Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall & Winter 2025
Lecture 4

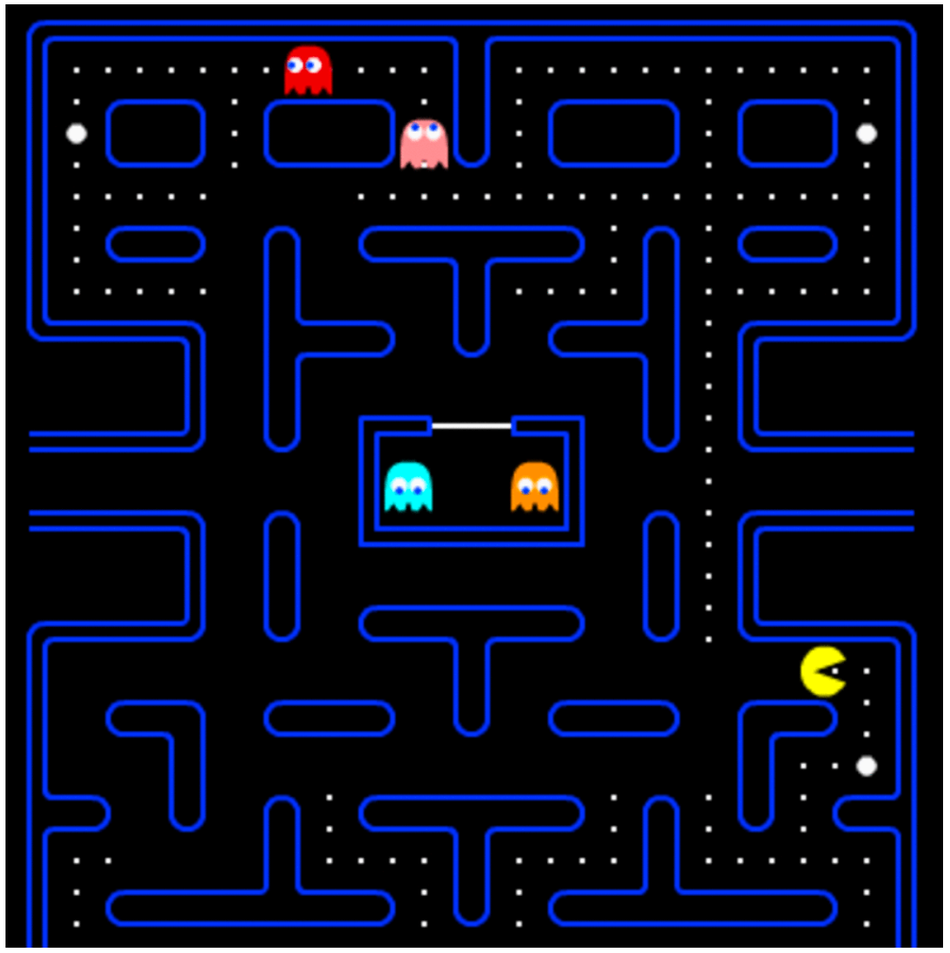
Search & Backtracking

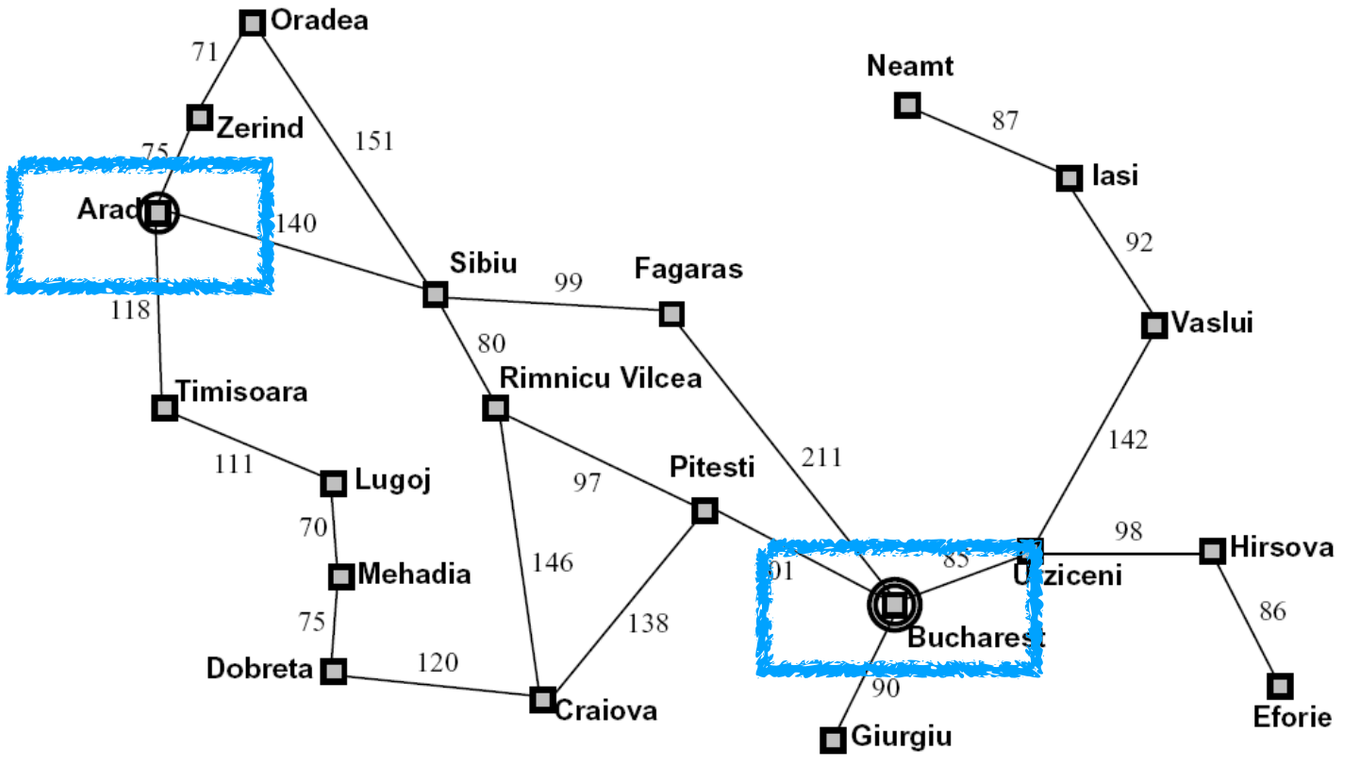
- Fundamental search
- Search with backtracking
- Search by simulation
- Take-home messages

Search & Backtracking

- Fundamental search
- Search with backtracking
- Search by simulation
- Take-home messages

Search Problem

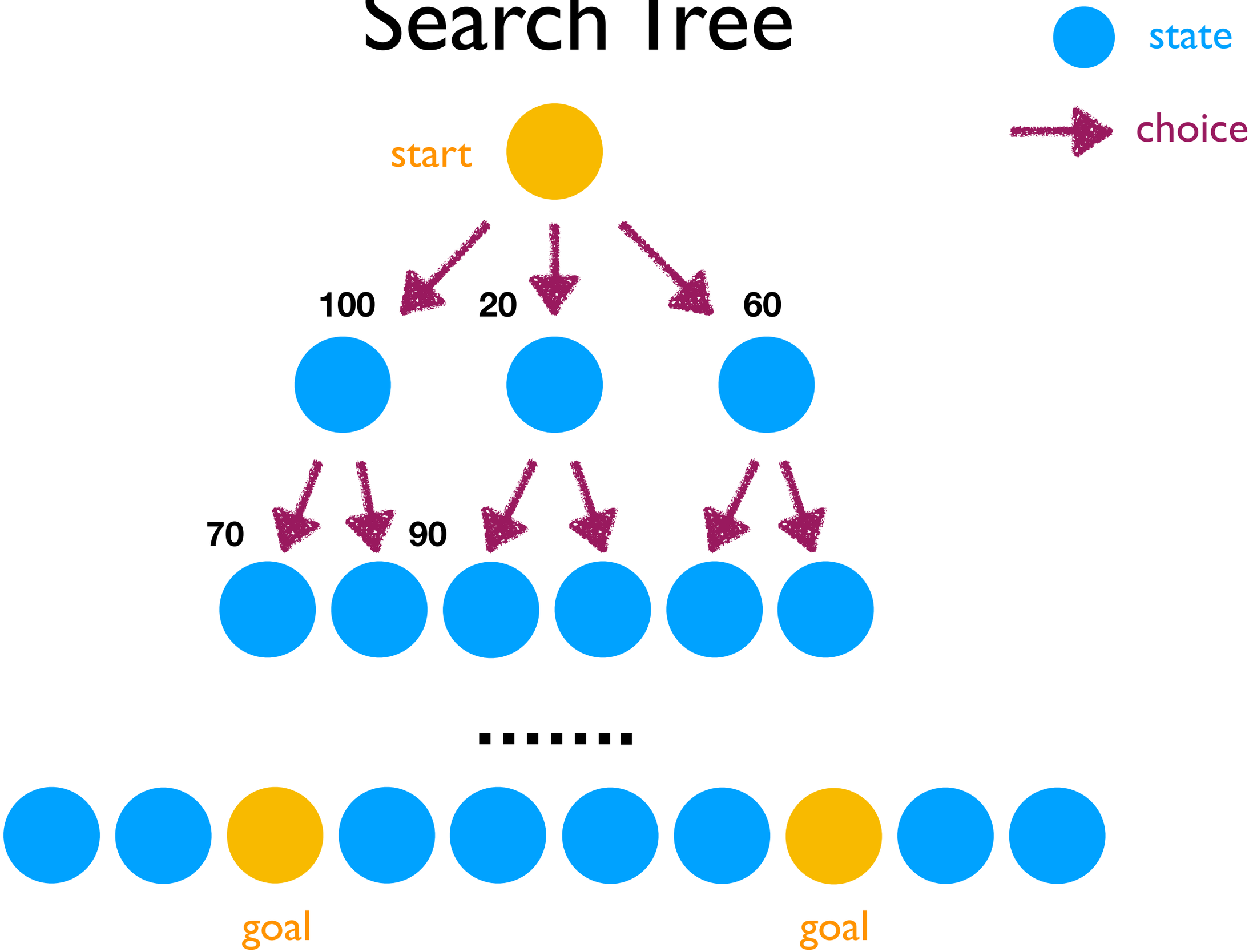




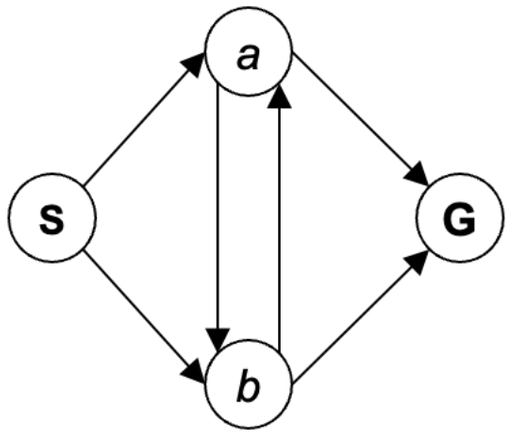
"Trees sprout up just about everywhere in computer science."

—— Donald Knuth

Search Tree



What is the Depth of the Search Tree for the Graph?



There may be a lot of repeated structures in a search tree.

Search Algorithms

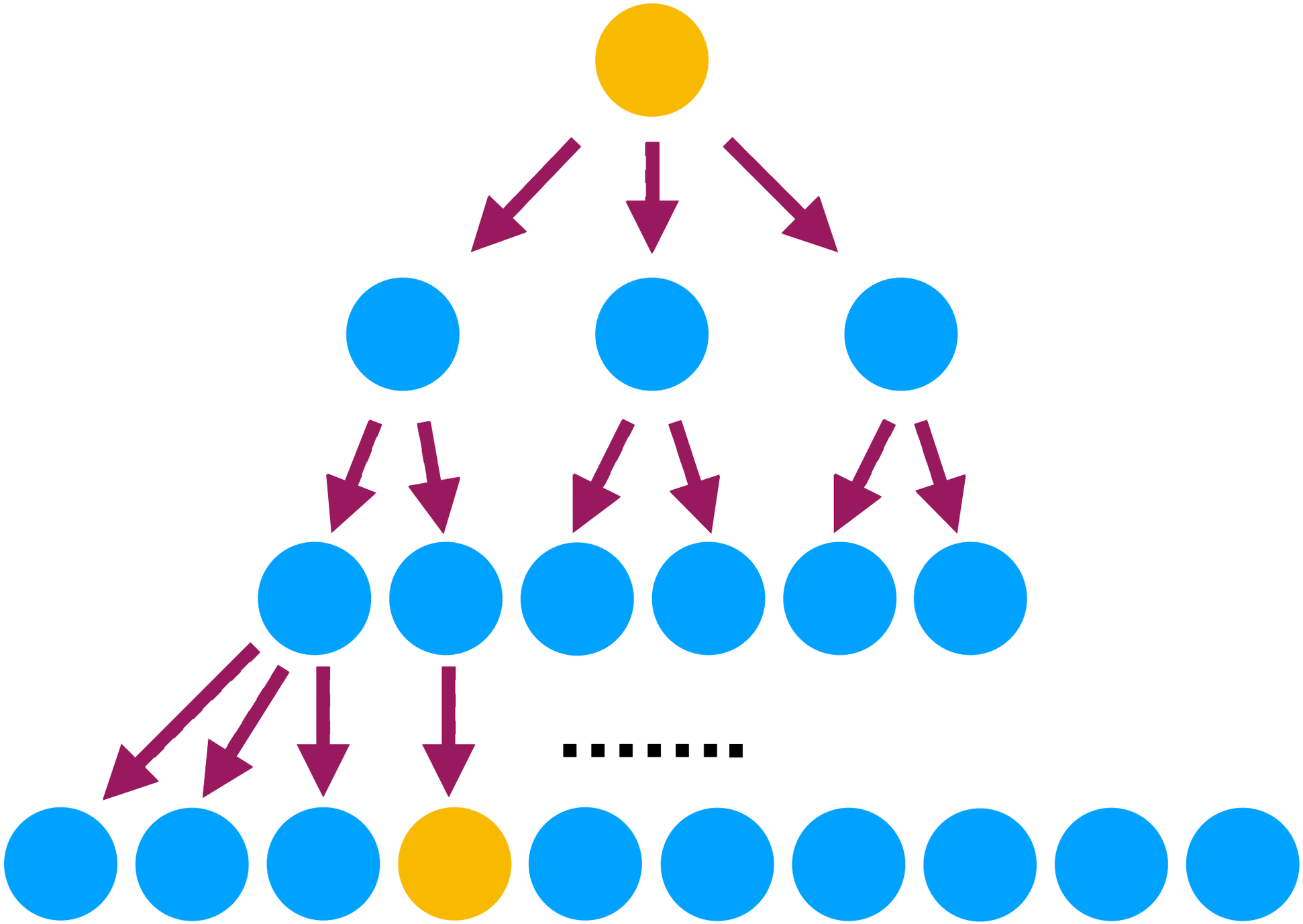
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
end
```

The strategy of choosing which leaf to expand is essential.

Breath-First Search

Notice: Temporally assume uniform cost for any action.

Breath-First Search (BFS)



Implemented with FIFO queue.

How Good is a Search Algorithm?

- Completeness:
  - Guarantee to find a solution if exists.
- Optimality:
  - Guarantee to find the least cost (largest utility) path.

caution: not about complexity

- Time complexity
- Space complexity

How Good is BFS?

- Completeness:
  - Yes

b : the maximum #edge on a node
d : the maximum depth of the search tree

- Optimality:
  - Yes when costs on all edges are equal
- Time complexity
  - $O(b^d)$

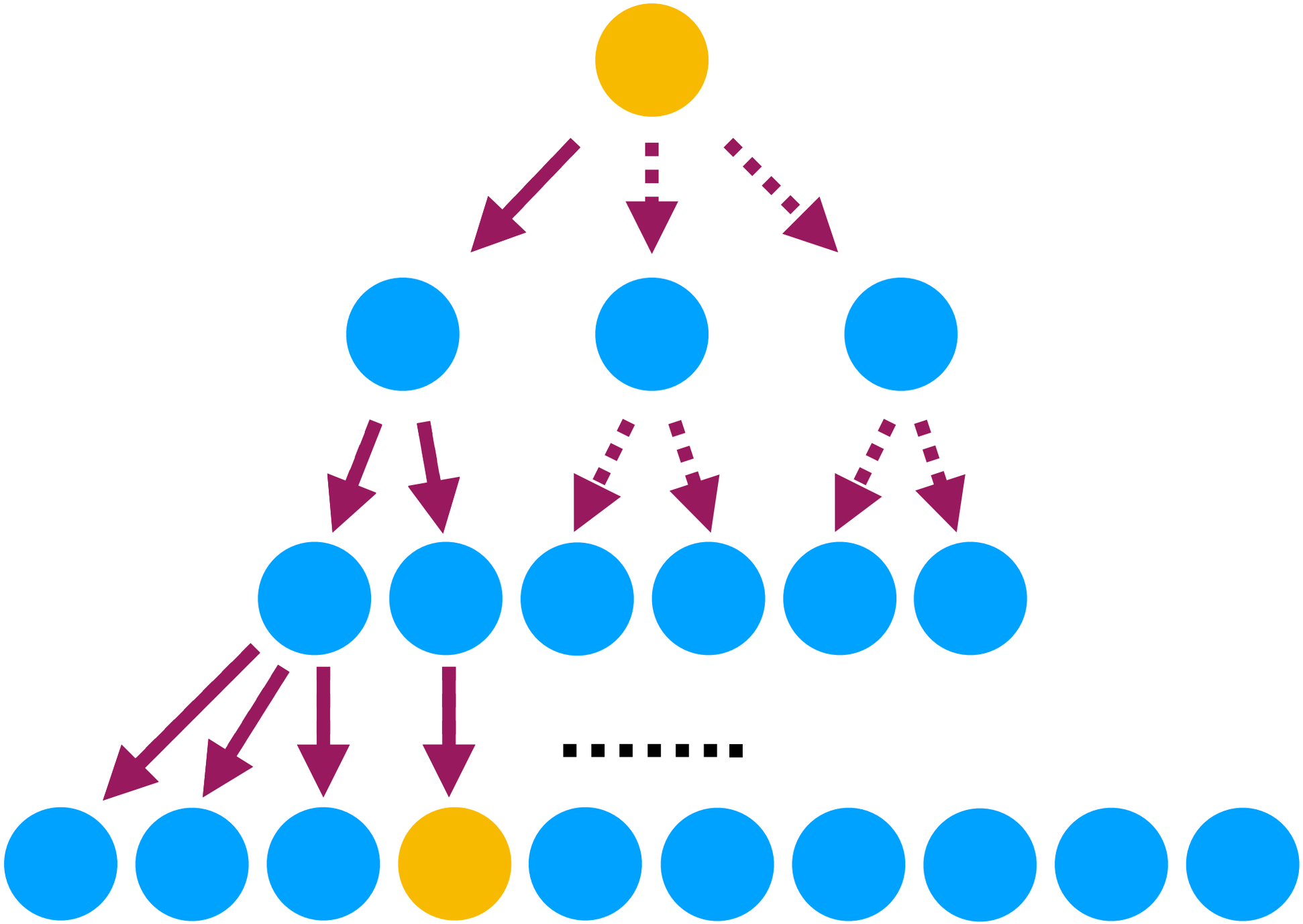
Difficult to improve if no more information about the search tree is given

- Space complexity
  - $O(b^d)$

Big issue for real-world application!
Anyway to improve?

Depth-First Search

Depth-First Search (DFS)



Implemented with LIFO queue (stack).

How Good is DFS?

- Completeness:
  - NO! (why?)

b : the maximum #edge on a node
d : the maximum depth of the search tree

- Optimality:
  - NO! (why?)

- Time complexity
  - $O(b^d)$

- Space complexity
  - $O(bd)$

Much more reasonable than BFS.
Making DFS more widely used in practice.

BFS vs. DFS

Can we get the benefits of both BFS and DFS?

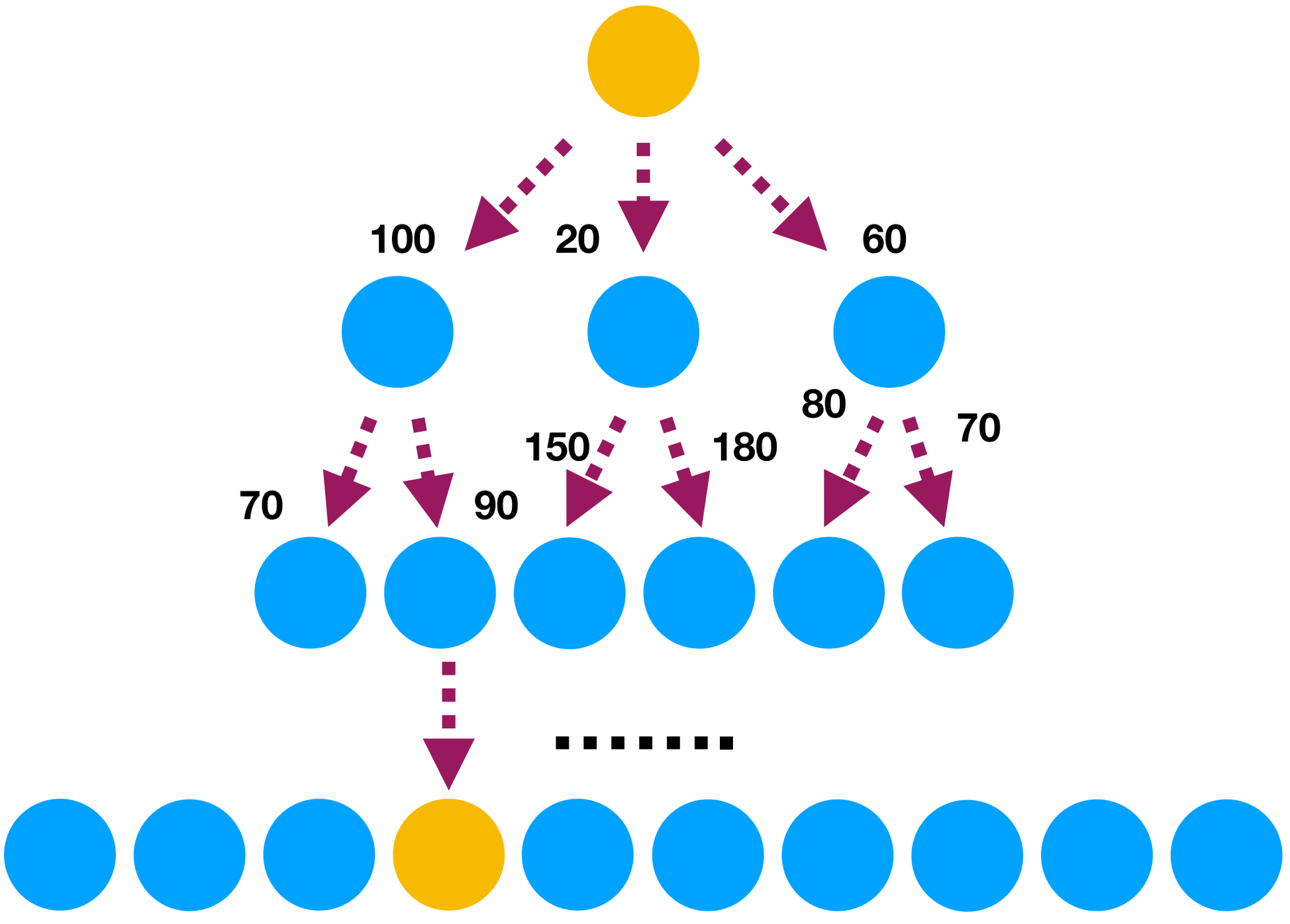
Iterative Deepening Search

- Run a DFS with depth limit 1. If no solution…
- Run a DFS with depth limit 2. If no solution…
- Run a DFS with depth limit 3. …

Why can we do this? Aren't many nodes searched repeatedly?

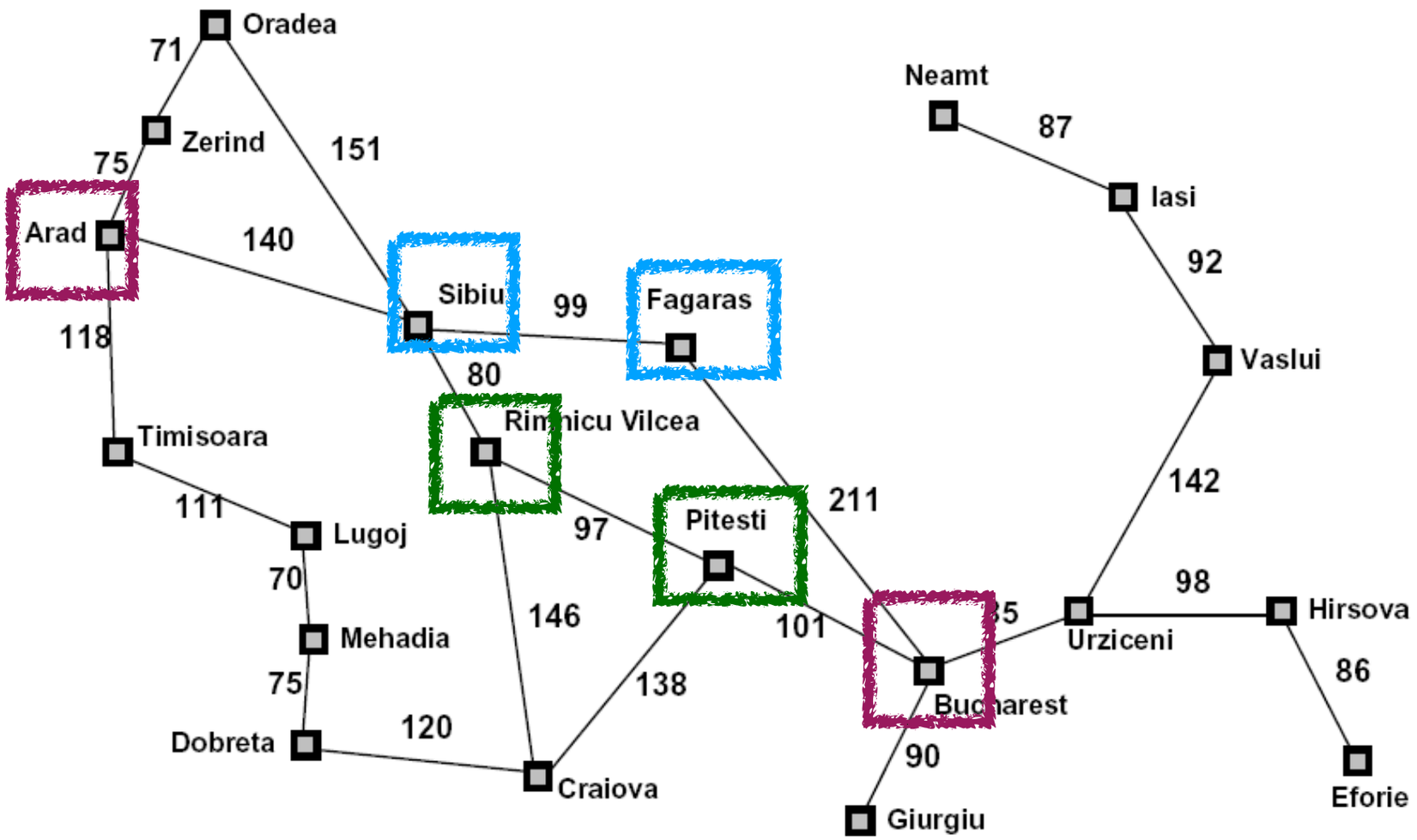
Actually, the time complexity is dominated by the latest DFS!

Cost-Sensitive Search



Neither BFS nor DFS works.

Best-First Search (Greedy Search)

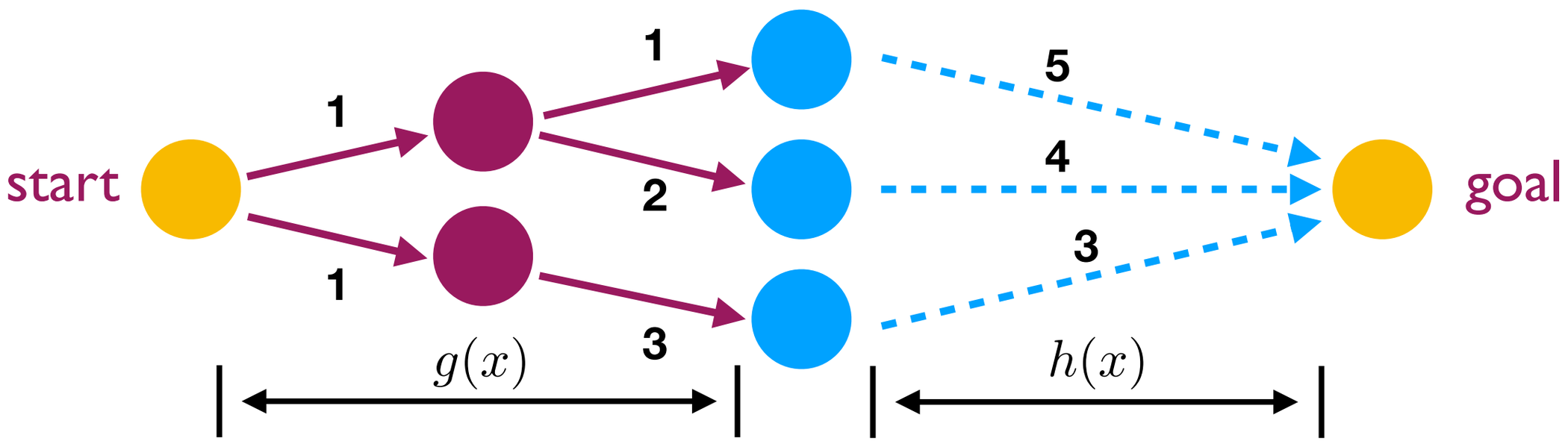


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Always expand the node with the best heuristic function value

As usual, greedy strategy can lead to sub-optimal results.

A* Search



- A* chooses one x to expand with minimum $f(x) = g(x) + h(x)$
- $g(x)$: known cost from the start to nodes.
- $h(x)$: heuristic to estimate the cost from nodes to the goal.

A* guarantees to find the optimal solution when $h(x)$ never underestimates the true cost.

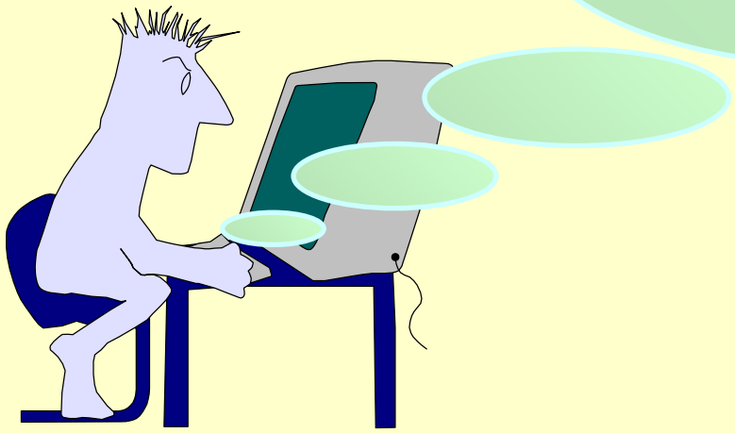
Search & Backtracking

- Fundamental search
- Search with backtracking
- Search by simulation
- Take-home messages

Rationale of the Backtracking Algorithms

A sure-fire way to find the answer to a problem is to make a list of all candidate answers, examine each, and following the examination of all or some of the candidates, declare the identified answer.

Suuuure — if the list is finite and it is possible to identify the answer following the examinations. AND, if there are not too many candidates.



Rationale of the Backtracking Algorithms

A sure-fire way to find the answer to a problem is to make a list of all candidate answers, examine each, and following the examination of all or some of the candidates, declare the identified answer.

Pruning in the search tree

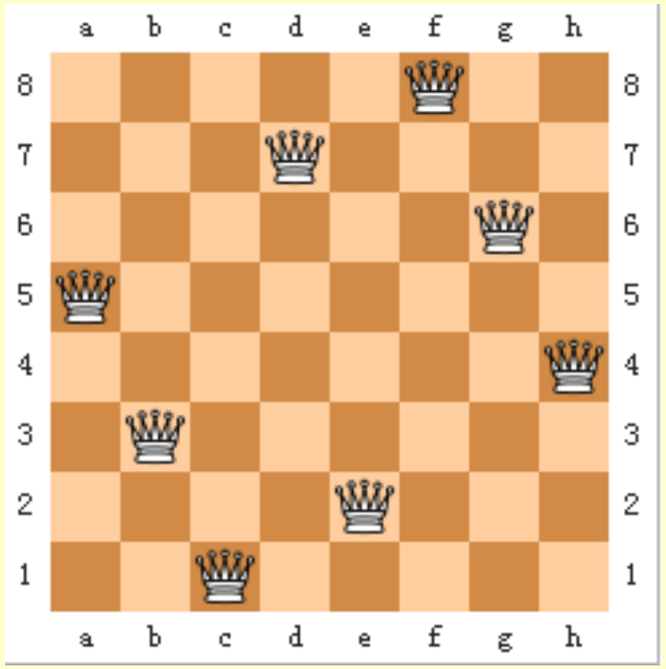
Backtracking enables us to eliminate the explicit examination of a large subset of the candidates while still guaranteeing that the answer will be found if the algorithm is run to termination.

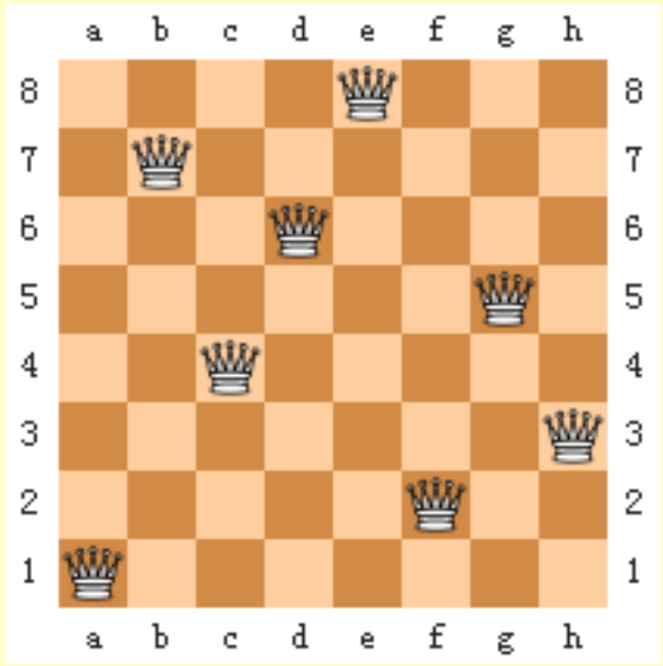
The basic idea is that suppose we have a partial solution (x_1, ... , x_i) where each $x_k \in S_k$ for $1 \le k \le i < n$. First we add $x_{i+1} \in S_{i+1}$ and check if (x_1, ... , x_i, x_{i+1}) satisfies the constraints. If the answer is "yes" we continue to add the next x, else we delete x_i and backtrack to the previous partial solution (x_1, ... , x_{i-1}).

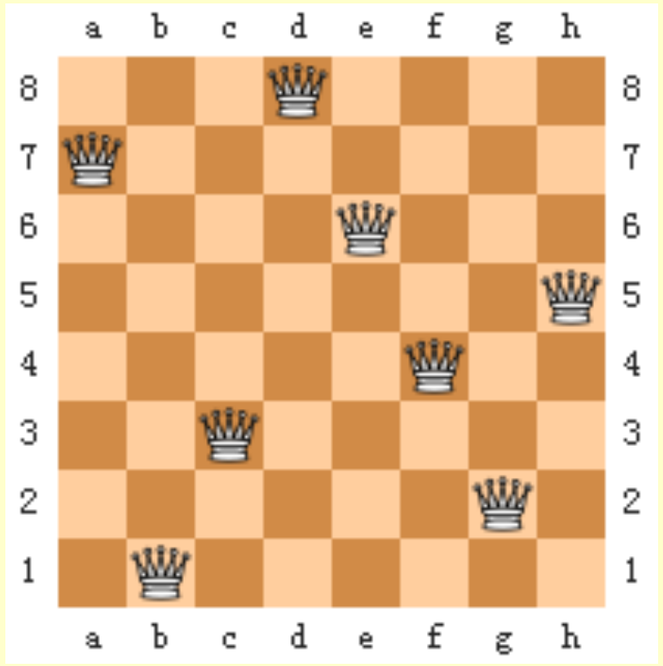
Eight Queens

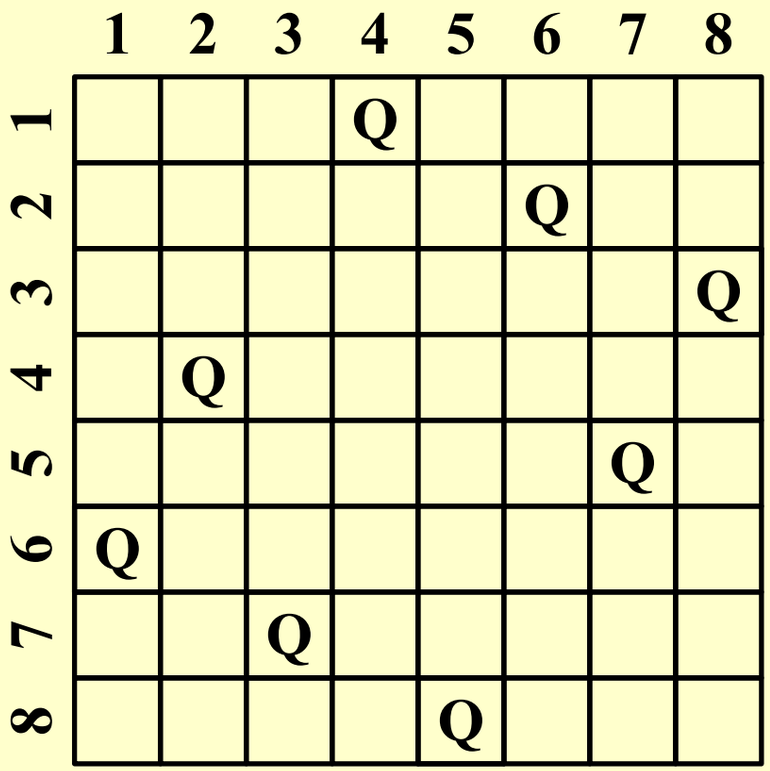
Find a placement of 8 queens on an 8 × 8 chessboard such that no two queens attack.

Two queens are said to attack iff they are in the same row, column, diagonal, or anti-diagonal of the chessboard.









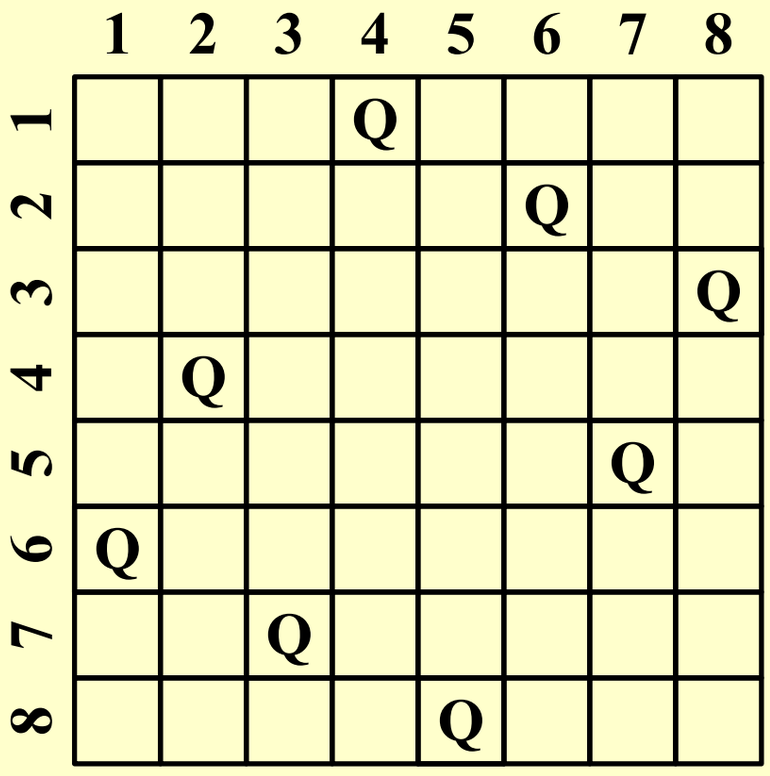
Q_i ::= queen in the i-th row

x_i ::= the column index in which Q_i is

Solution = $(x_1, x_2, \ldots , x_8)$

= (4, 6, 8, 2, 7, 1, 3, 5)

Constraints: ① $S_i = \{ 1,2,3,4,5,6,7,8 \}$ for $1 \le i \le 8$



Q_i ::= queen in the i-th row

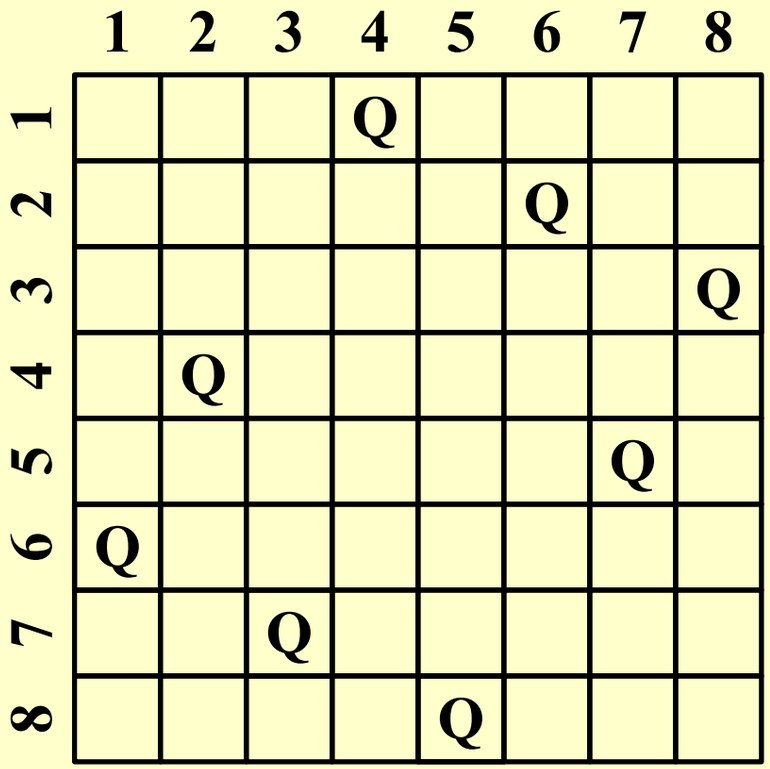
x_i ::= the column index in which Q_i is

Solution = ($x_1, x_2, \ldots, x_8$)

= (4, 6, 8, 2, 7, 1, 3, 5)

This implies 8^8 candidates in the solution space.

Constraints: ① $S_i = \{ 1,2,3,4,5,6,7,8 \}$ for $1 \le i \le 8$



Q_i ::= queen in the i-th row

x_i ::= the column index in which Q_i is

S

This implies
that the solution must
be a permutation of 1, 2, ... , 8.
Thus the number of candidates
in the solution space
is reduced to 8!.

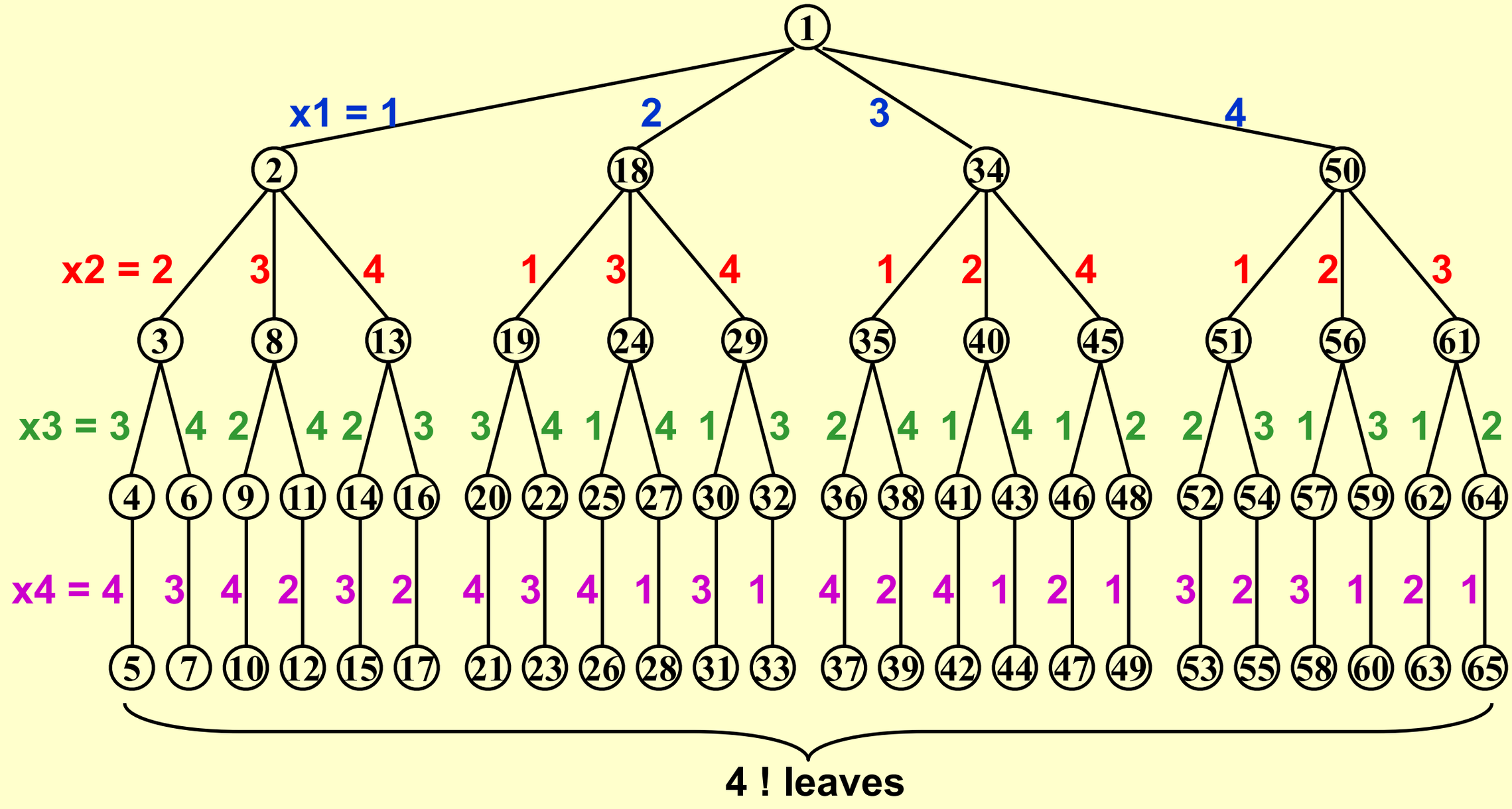
Constraints: ① $S_i = \{ 1,2,3,4,5,6,7,8 \}$ for $1 \le i \le 8$

② $x_i \neq x_j$ if $i \neq j$ ③ $(x_i - x_j) / (i - j) \neq \pm 1$

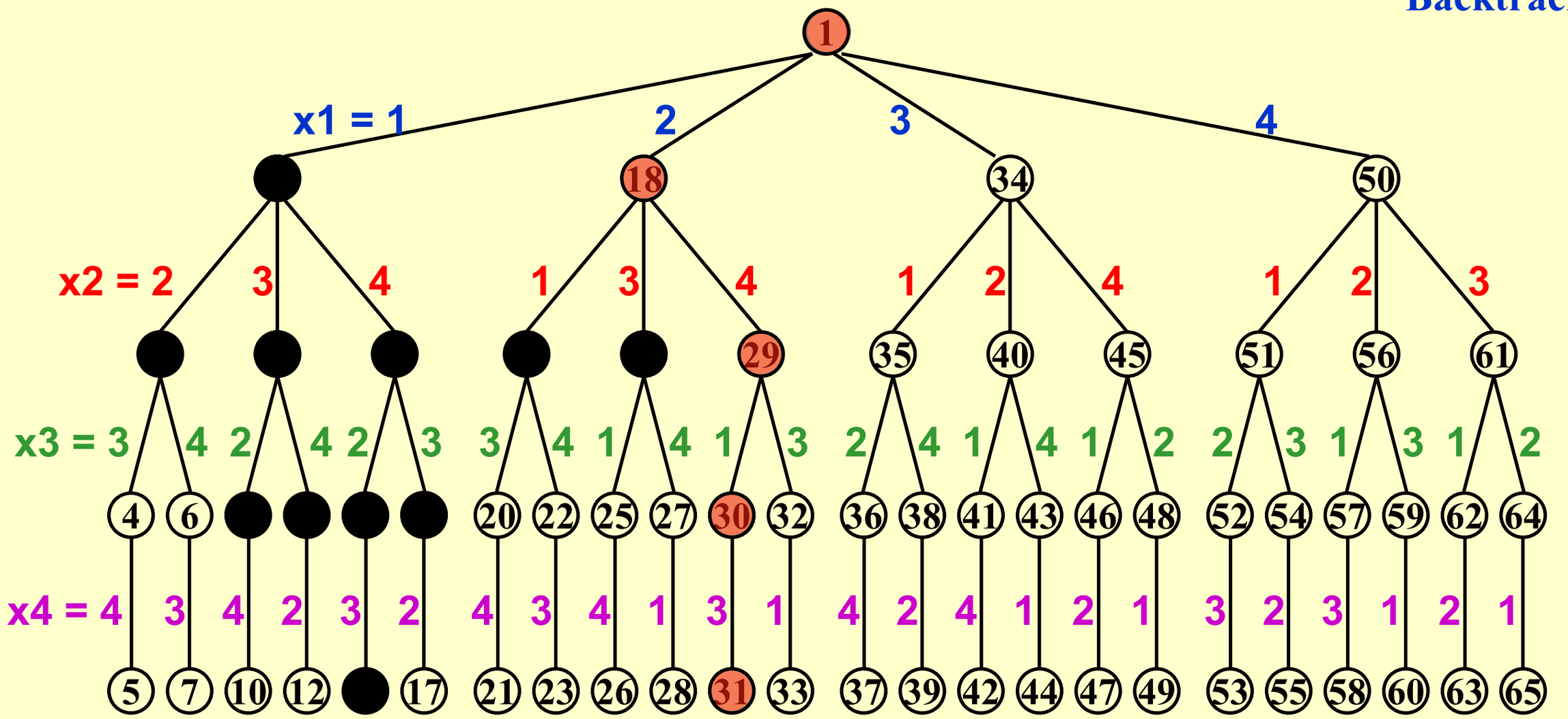
For the problem with n queens,
there are $n!$ candidates
in the solution space.

Method: Take the problem of 4 queens as an example

Step 1: Construct a game tree

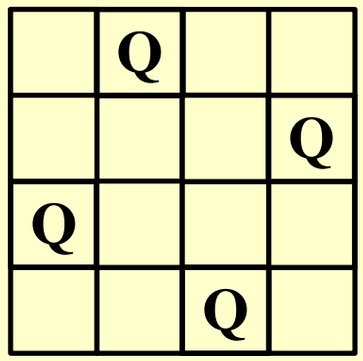


Each path from the root to a leaf defines an element of the solution space.



Step 2: Perform a depth-first search (post-order traversal) to examine the paths

(2, 4, 1, 3)



Note: No tree is actually constructed. The game tree is just an abstract concept.

The Turnpike Reconstruction Problem

Given N points on the x-axis with coordinates $x_1 < x_2 < \ldots < x_N$. Assume that $x_1 = 0$. There are $N(N-1)/2$ distances between every pair of points.

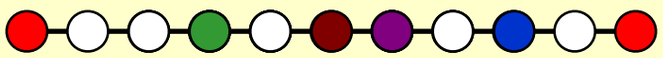
Given $N(N-1)/2$ distances. **Reconstruct a point set** from the distances.

〖Example〗 Given D = { 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10 }

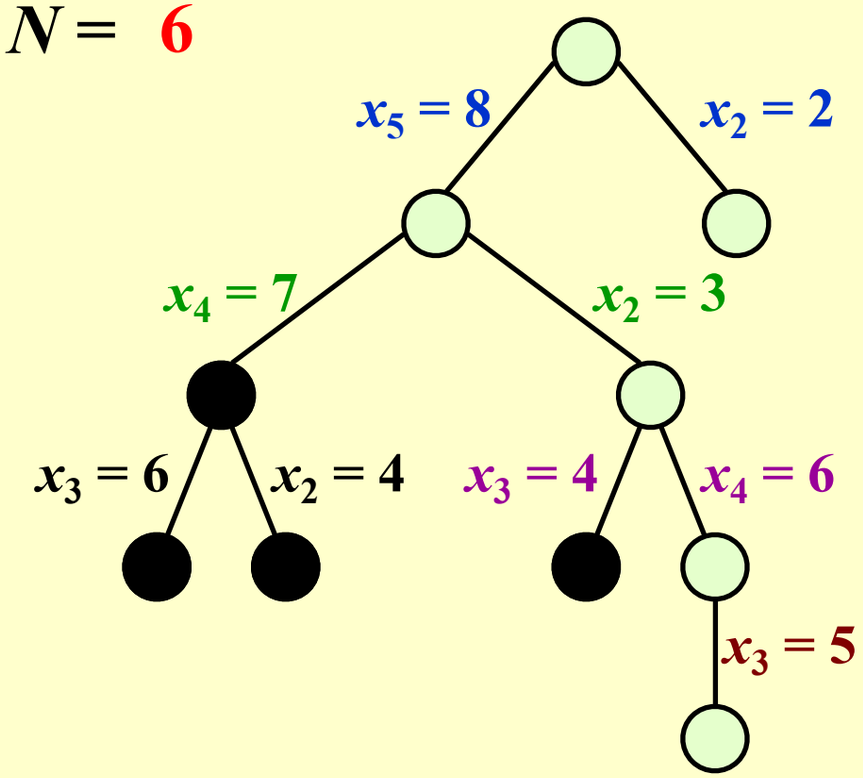
Step 1: $N(N-1)/2 = 15$ implies $N =$ 6

Step 2: $x_1 = 0$ and $x_6 = 10$

Step 3: find the next largest distance and check



(0, 3, 5, 6, 8, 10)



```
bool Reconstruct ( DistType X[ ], DistSet D, int N, int left, int right )
{ /* X[1]...X[left-1] and X[right+1]...X[N] are solved */
    bool Found = false;
    if ( Is_Empty( D ) )
        return true; /* solved */
    D_max = Find_Max( D );
    /* option 1：  X[right] = D_max */
    /* check if |D_max-X[i]|∈D is true for all X[i]'s that have been solved */
    OK = Check( D_max, N, left, right ); /* pruning */
    if ( OK ) { /* add X[right] and update D */
        X[right] = D_max;
        for ( i=1; i<left; i++ )  Delete( |X[right]-X[i]|, D);
        for ( i=right+1; i<=N; i++ )  Delete( |X[right]-X[i]|, D);
        Found = Reconstruct ( X, D, N, left, right-1 );
        if ( !Found ) { /* if does not work, undo */
            for ( i=1; i<left; i++ )  Insert( |X[right]-X[i]|, D);
            for ( i=right+1; i<=N; i++ )  Insert( |X[right]-X[i]|, D);
        }
    }
    /* finish checking option 1 */
```

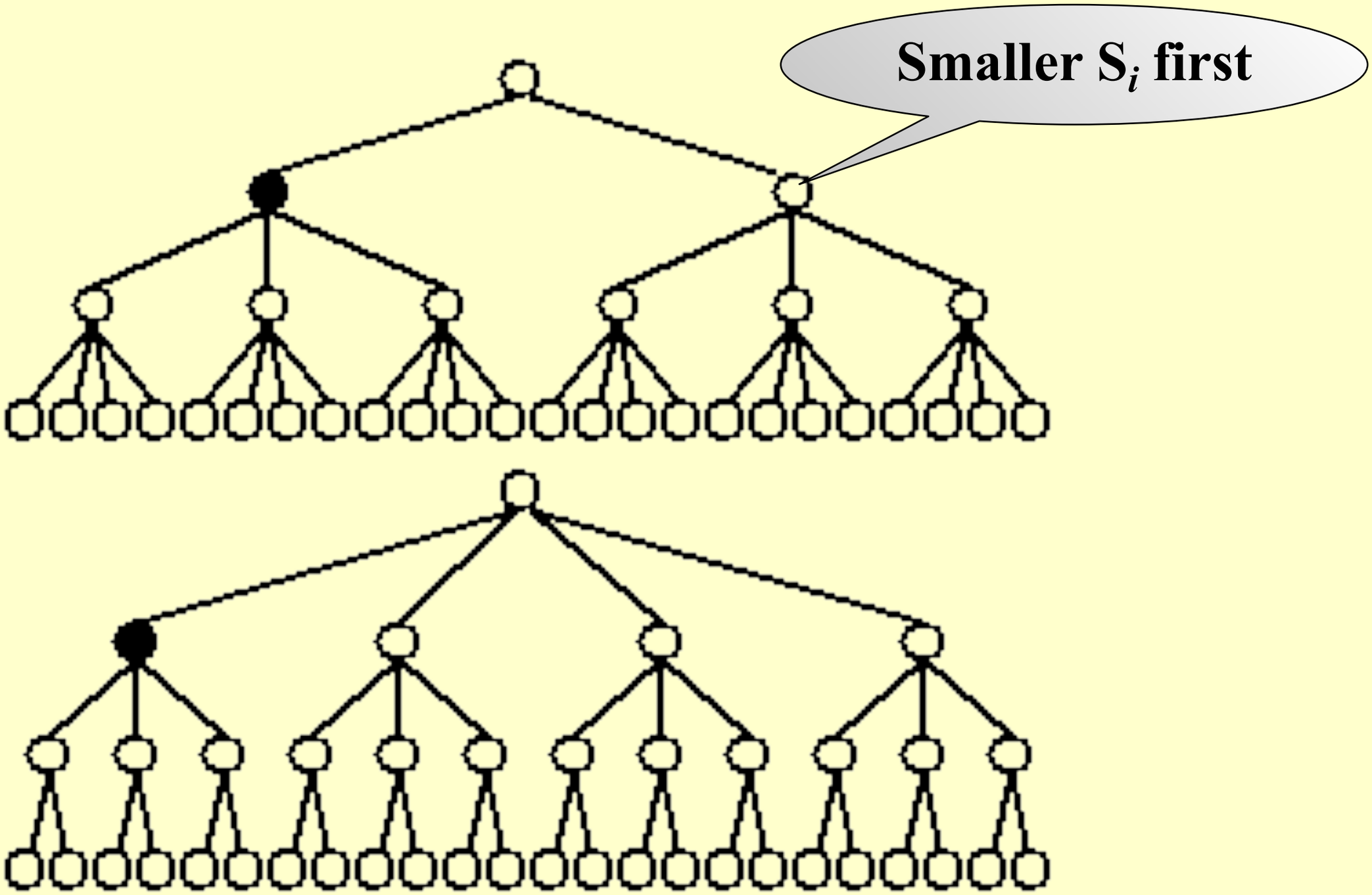
```
if ( !Found ) { /* if option 1 does not work */
    /* option 2: X[left] = X[N]-D_max */
    OK = Check( X[N]-D_max, N, left, right );
    if ( OK ) {
        X[left] = X[N] – D_max;
        for ( i=1; i<left; i++ )  Delete( |X[left]-X[i]|, D);
        for ( i=right+1; i<=N; i++ )  Delete( |X[left]-X[i]|, D);
        Found = Reconstruct (X, D, N, left+1, right );
        if ( !Found ) {
            for ( i=1; i<left; i++ ) Insert( |X[left]-X[i]|, D);
            for ( i=right+1; i<=N; i++ ) Insert( |X[left]-X[i]|, D);
        }
    }
    /* finish checking option 2 */
  } /* finish checking all the options */

  return Found;
}
```

A Template

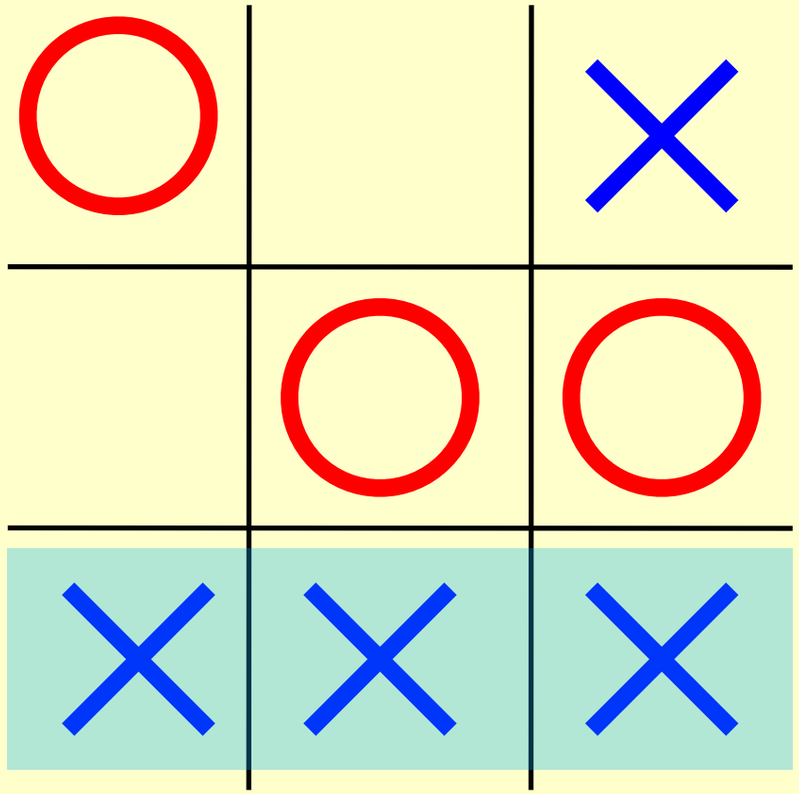
```
bool Backtracking ( int i )
{   Found = false;
    if ( i > N )
        return true; /* solved with (x1, …, xN) */
    for ( each xi ∈ Si ) {
        /* check if satisfies the restriction R */
        OK = Check((x1, …, xi) , R ); /* pruning */
        if ( OK ) {
            Count xi in;
            Found = Backtracking( i+1 );
            if ( !Found )
                Undo( i ); /* recover to (x1, …, xi-1) */
        }
        if ( Found ) break;
    }
    return Found;
}
```

When different S_i's have different sizes



Games – *how did AlphaGo win*

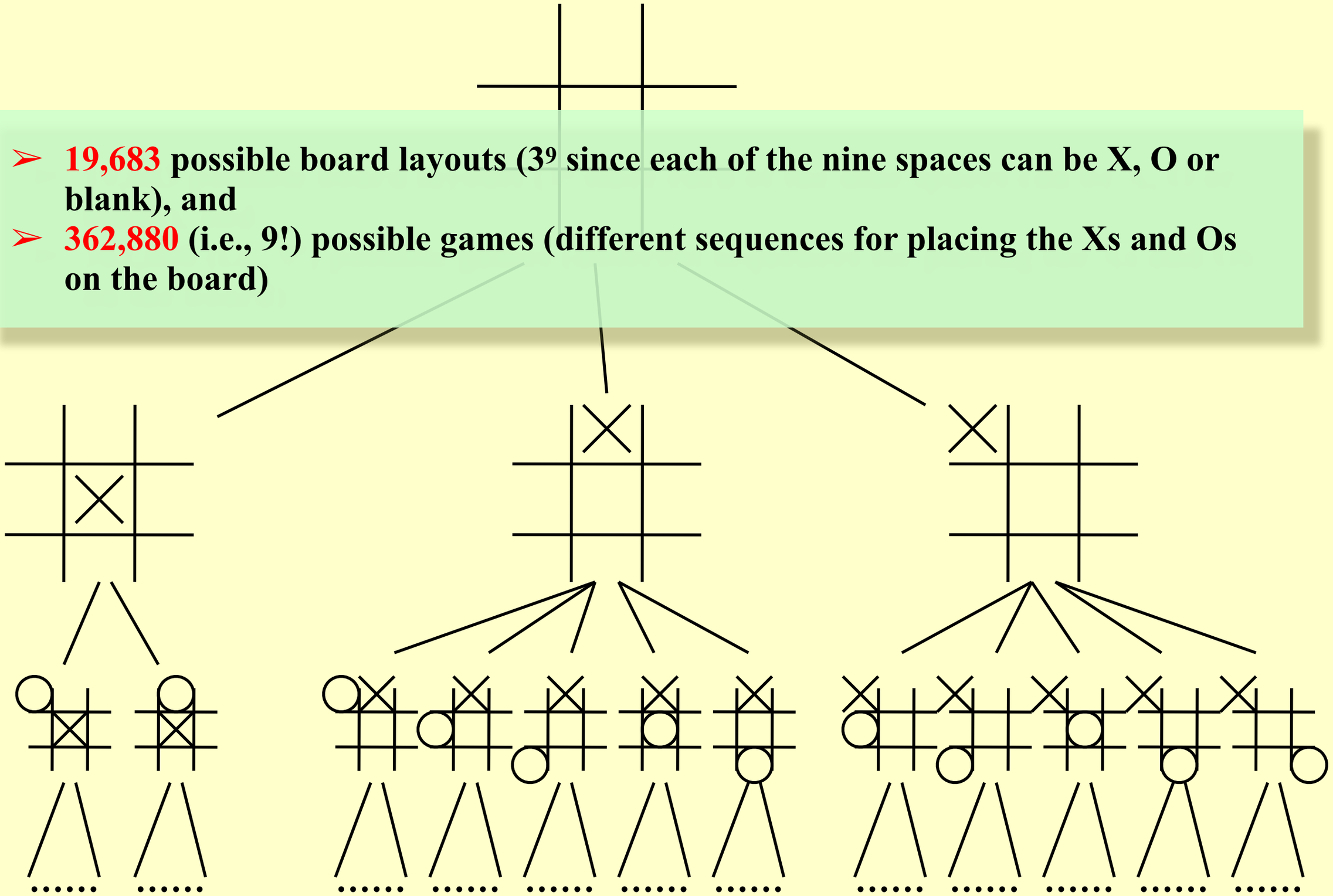
Tic-tac-toe



The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

Tic-tac-toe

- **19,683 possible board layouts (3^9 since each of the nine spaces can be X, O or blank), and**
- **362,880 (i.e., 9!) possible games (different sequences for placing the Xs and Os on the board)**

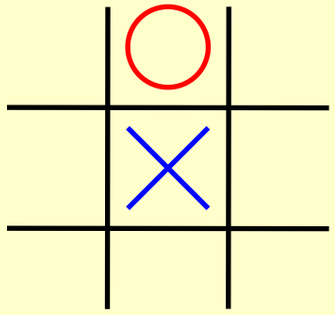


Tic-tac-toe: Minimax Strategy

Use an evaluation function to quantify the "goodness" of a position. For example:

$$f(P) = W_{Computer} - W_{Human}$$

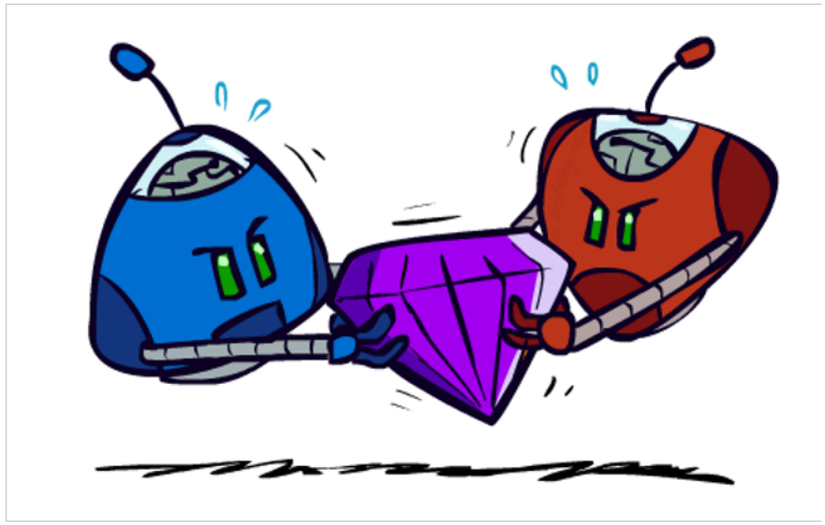
where W is the number of potential wins at position P.



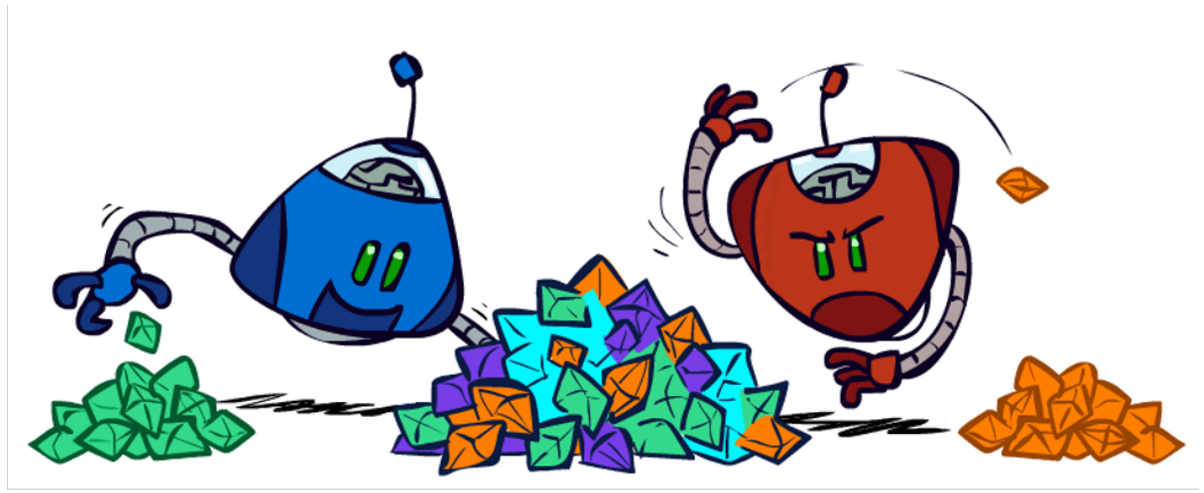
$$f(P) = 6 - 4 = 2$$

The human is trying to *minimize* the value of the position P, while the computer is trying to *maximize* it.

Two-Player Zero-Sum Game



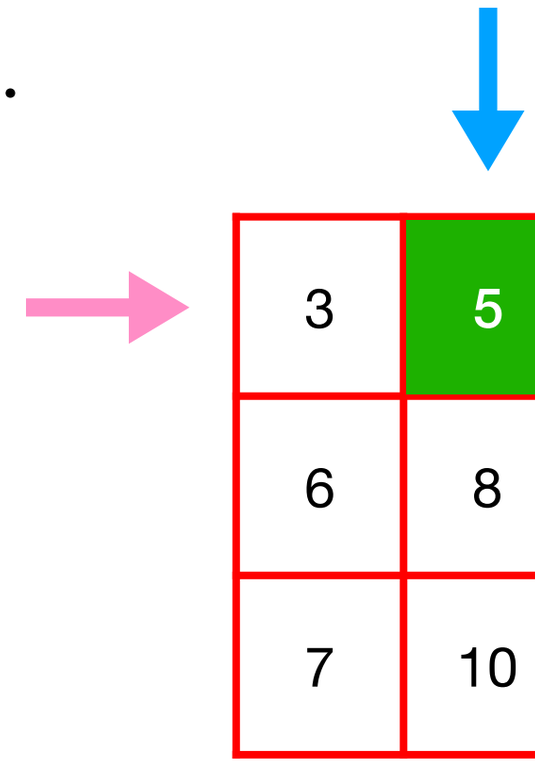
vs.



In zero-sum games, the utility functions of the two players are coupled:
How much one wins equals to how much the other loses: competitive.
Can the players be cooperative?

The Simplest Formulation: Two-Step Game

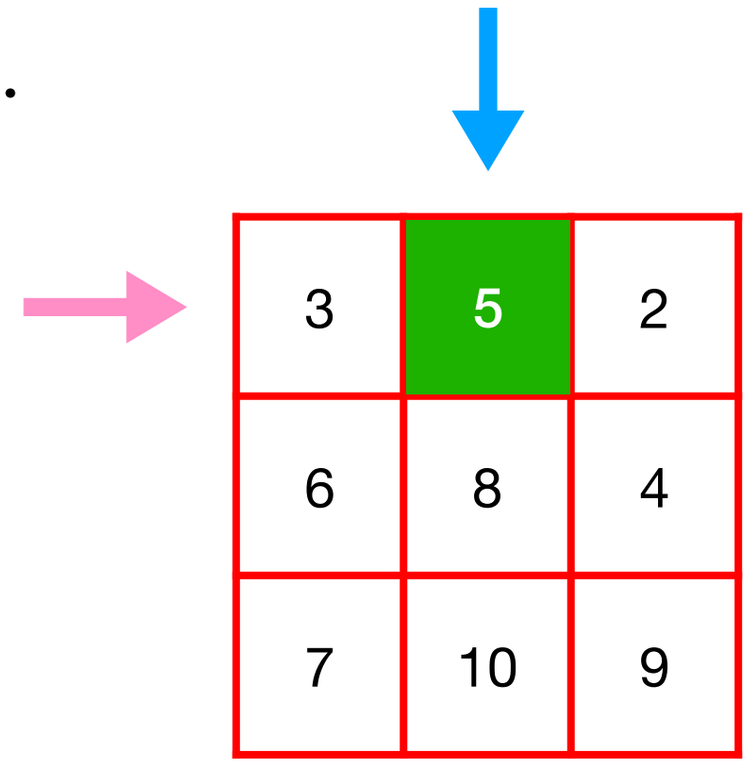
- Two step game: Alice chooses row i , then Bob chooses column j
- Outcome: Alice loses (Bob wins) the utility in entry i, j zero-sum
- A MinMax Game.



$$U(i^*, j^*) = \min_i \max_j U(i, j)$$

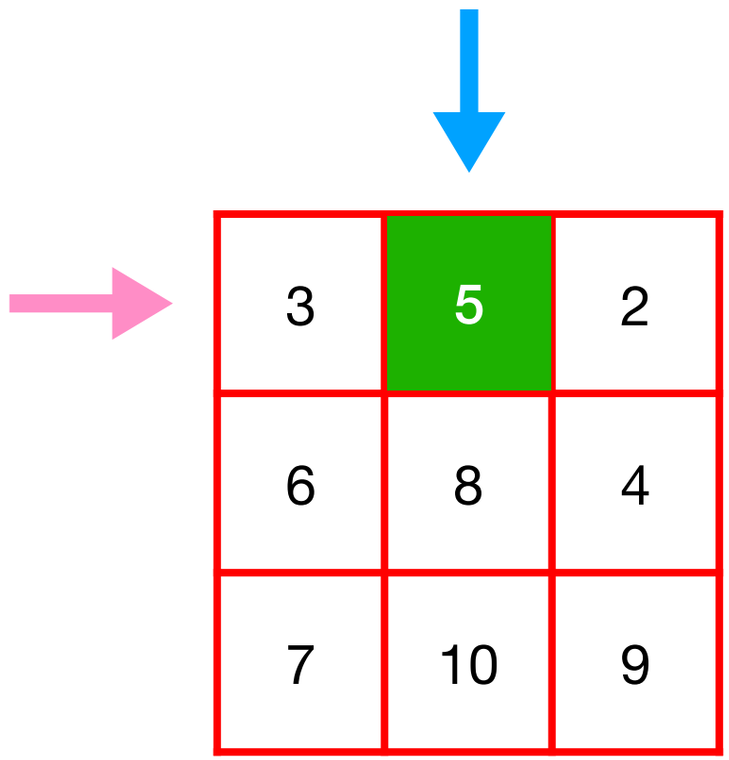
The Simplest Formulation: Two-Step Game

- Two step game: Bob chooses row j, then Alice chooses column i
- Outcome: Alice loses (Bob wins) the utility in entry i, j
- A MaxMin Game.



$$U(i^*, j^*) = \max_j \min_i U(i, j)$$

Is MinMax equivalent to MaxMin?



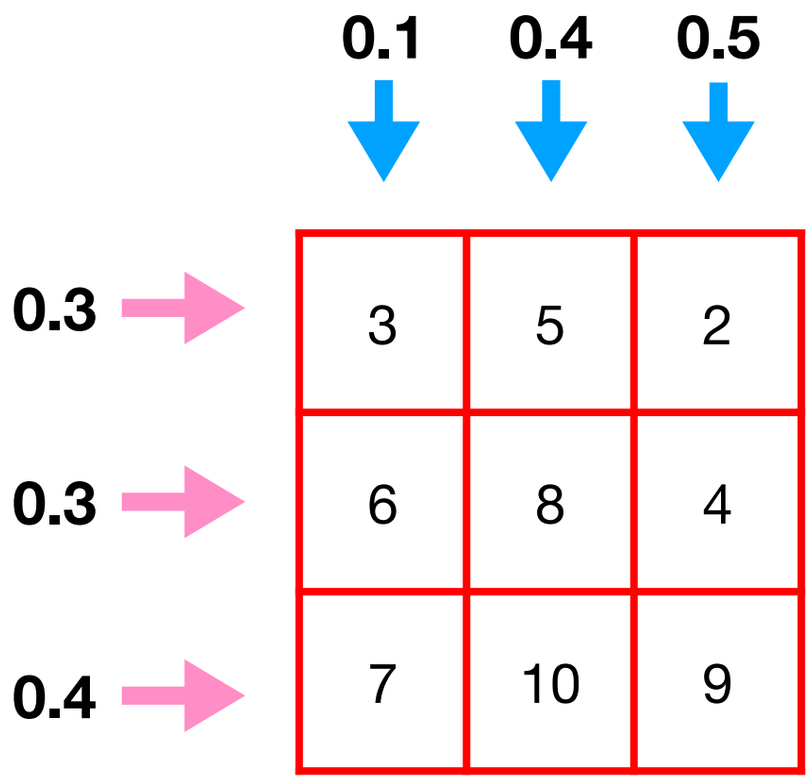
Does Alice (Bob) lose (win) the same utility in MinMax and MaxMin games?

Theorem:

$$\max_j \min_i U(i,j) \leq \min_i \max_j U(i,j)$$

Moving first is always worse!

Deterministic vs. Stochastic Strategy



Mixed (Stochastic) strategy: choosing a distribution over actions instead of a single action (pure or deterministic strategy)

Von Neumann's Minimax Theorem:

$$\min_{P_i} \max_{P_j} \mathbb{E}_{P_i,P_j}\left[U(i,j)\right] = \max_{P_j} \min_{P_i} \mathbb{E}_{P_i,P_j}\left[U(i,j)\right]$$

MinMax and MaxMin Games are equivalent for mixed strategy!

Nash Equilibrium

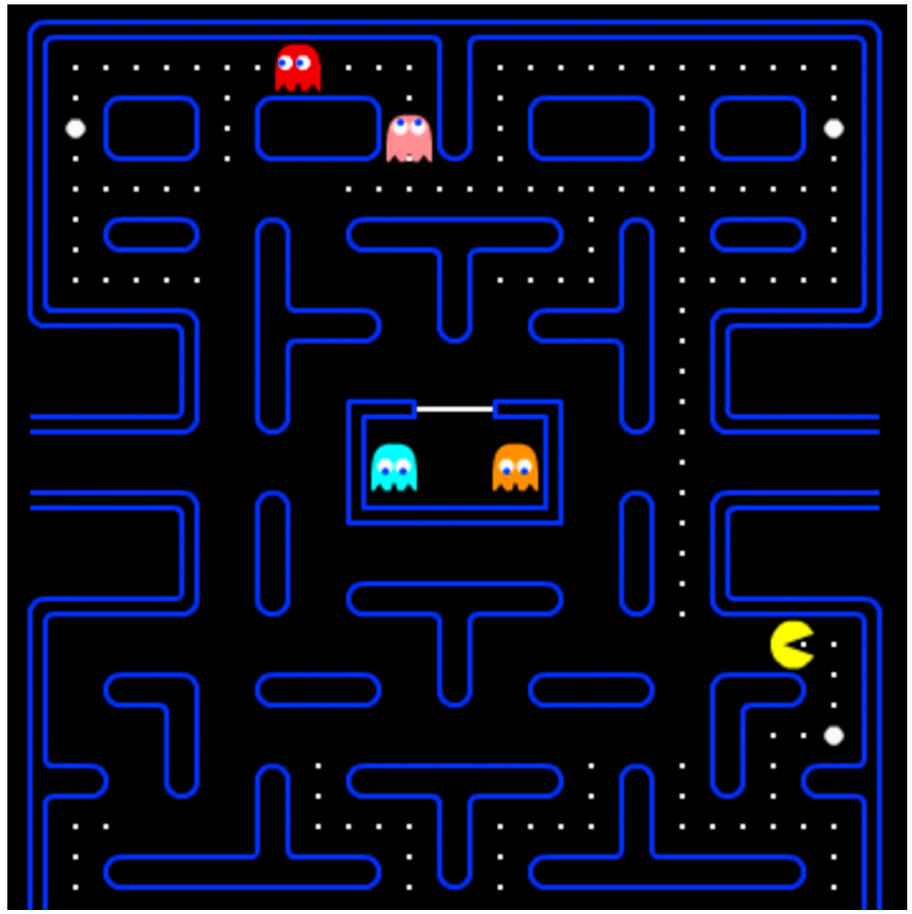
$$\min_{P_i} \max_{P_j} \mathbb{E}_{P_i, P_j} \left[U(i, j) \right] = \max_{P_j} \min_{P_i} \mathbb{E}_{P_i, P_j} \left[U(i, j) \right]$$

- Strong duality: the solution pair P_i^*, P_j^* achieving this equation is called the saddle point of the two-step zero-sum game.
- This is also the Nash equilibrium of the game.

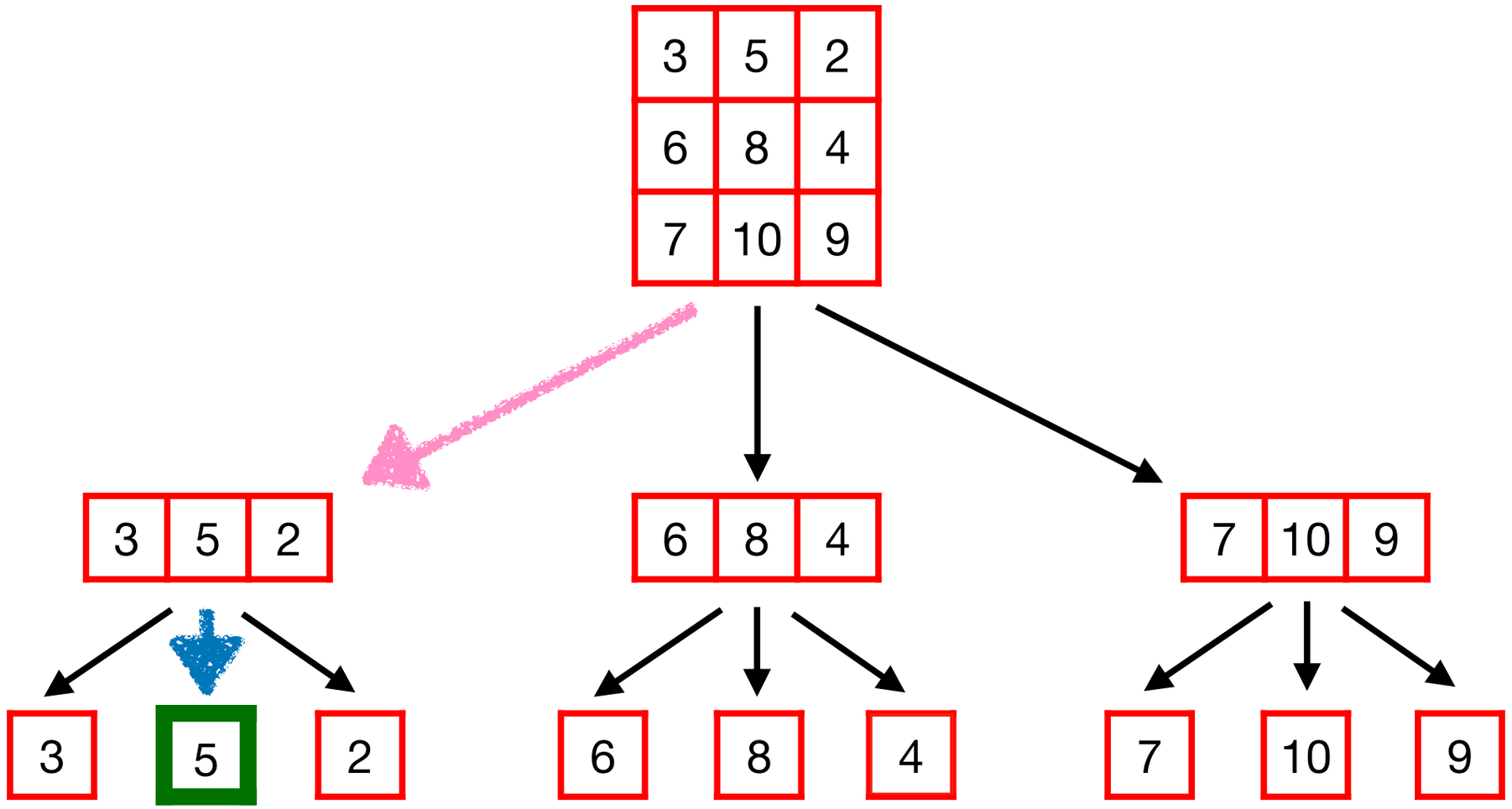
Under Nash equilibrium, changing the strategy for any player herself would not be a good idea.
But Nash equilibrium does not mean optimal strategy!
You know the prisoner's dilemma.

Multi-Step Zero-Sum Game

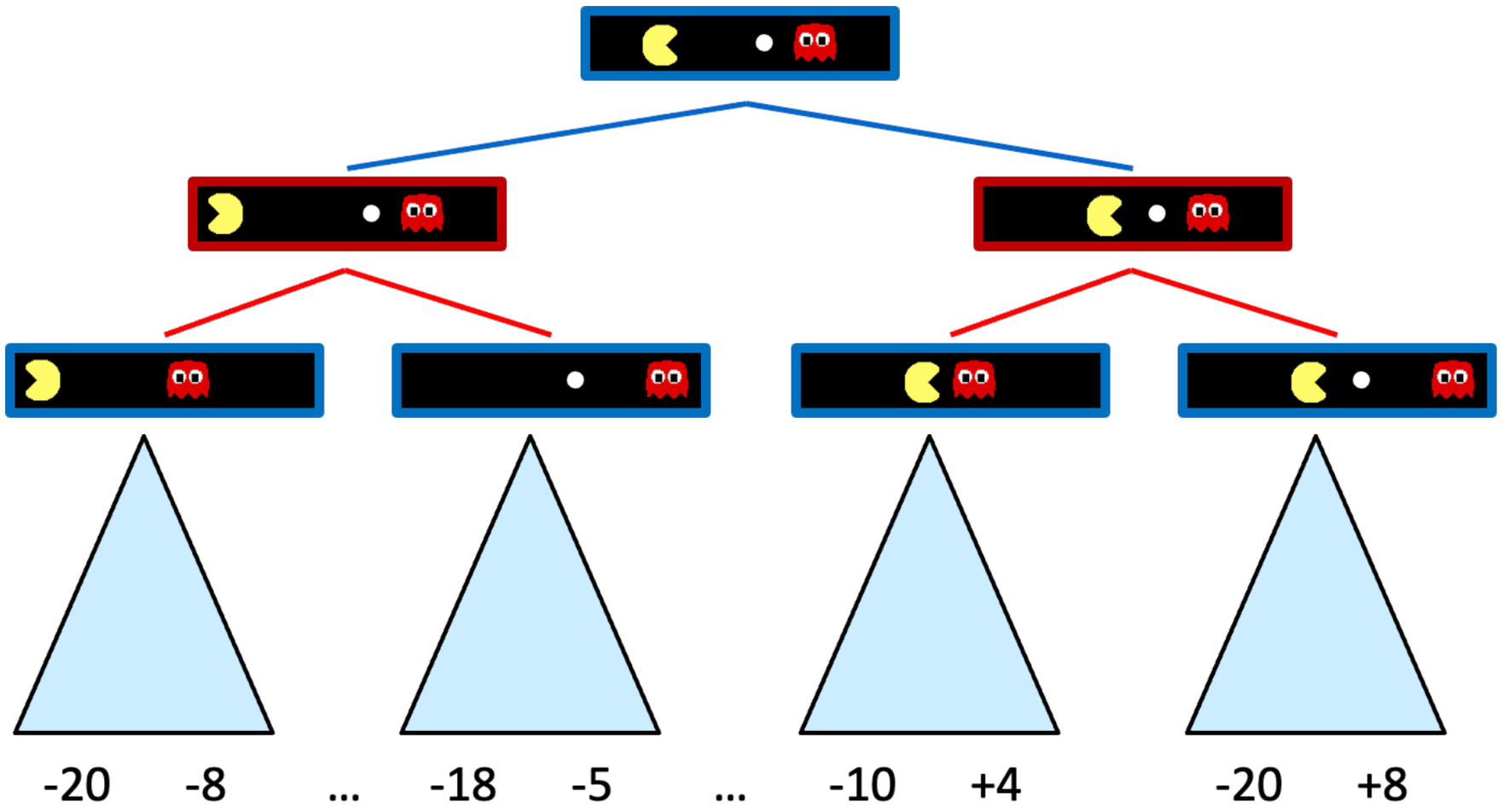
3	5	2
6	8	4
7	10	9



Game Tree of Two-Step Games



Multi-Step Zero-Sum Game



Can also be modeled as a search tree.

Lesson learned from two-step game:
The players should play rationally: use min and max strategies.
The players should look ahead when making decisions.

Min and Max Strategy

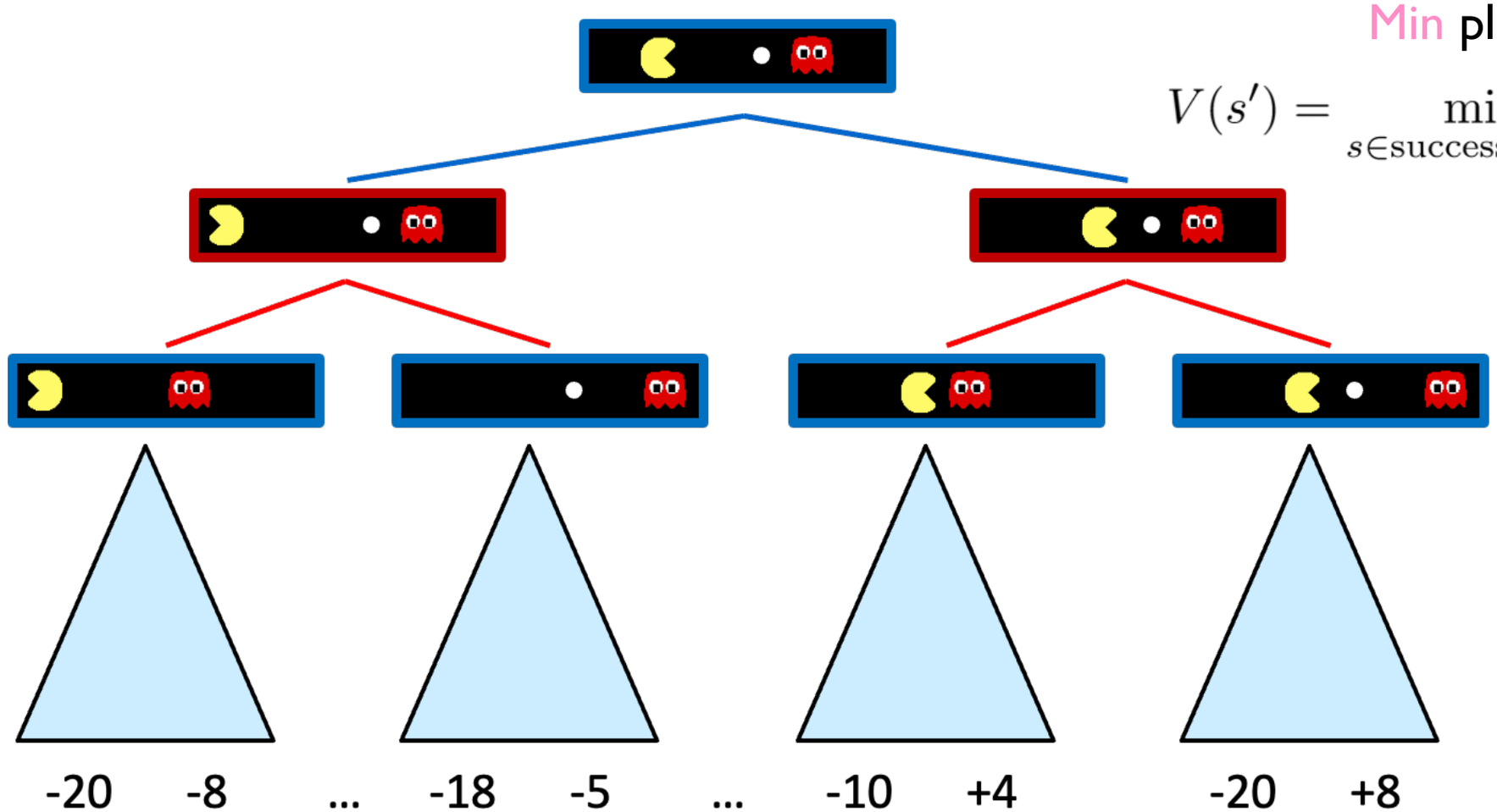
The game is deterministic and complete:
The search tree is known to the players.

Max player

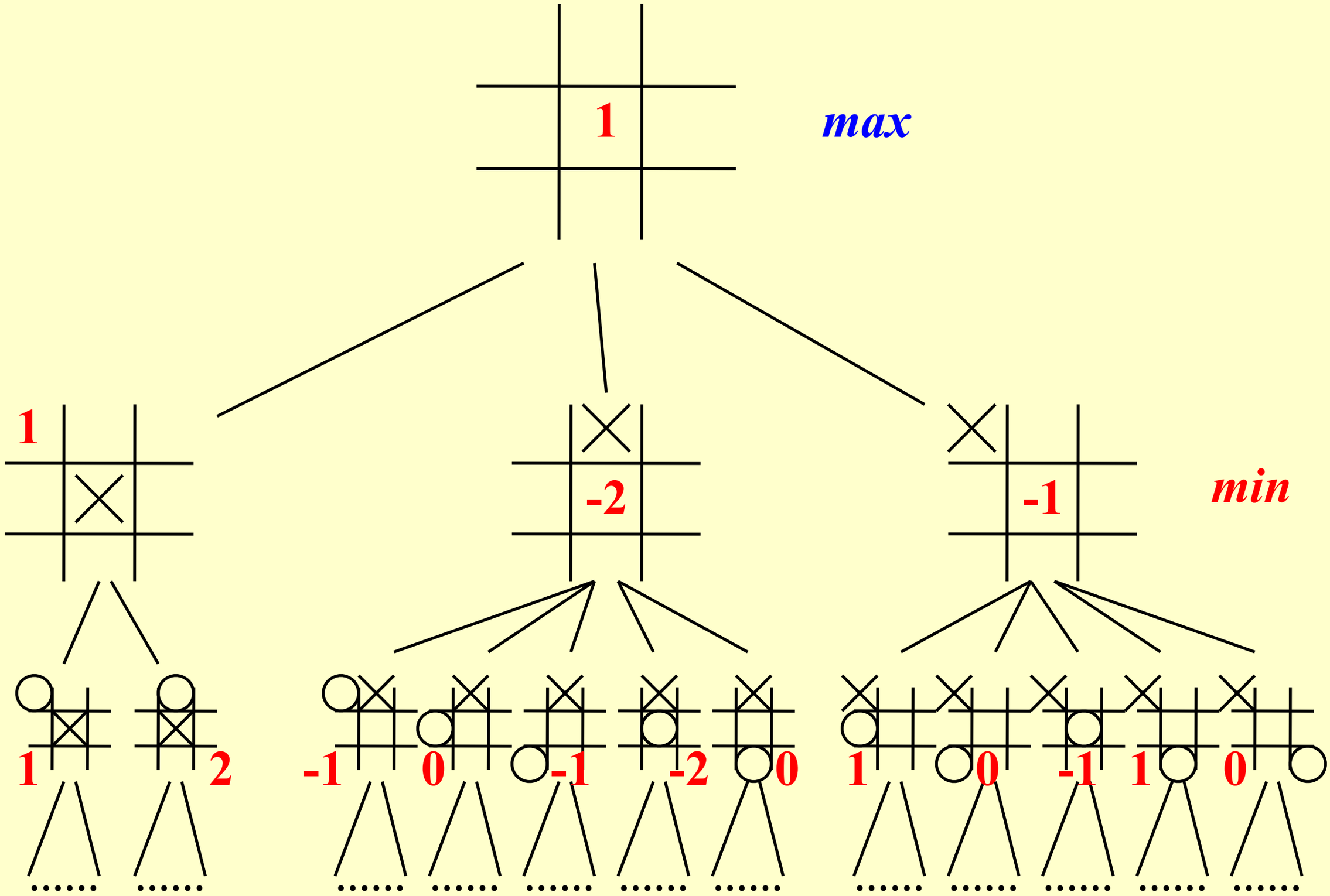
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

Min player

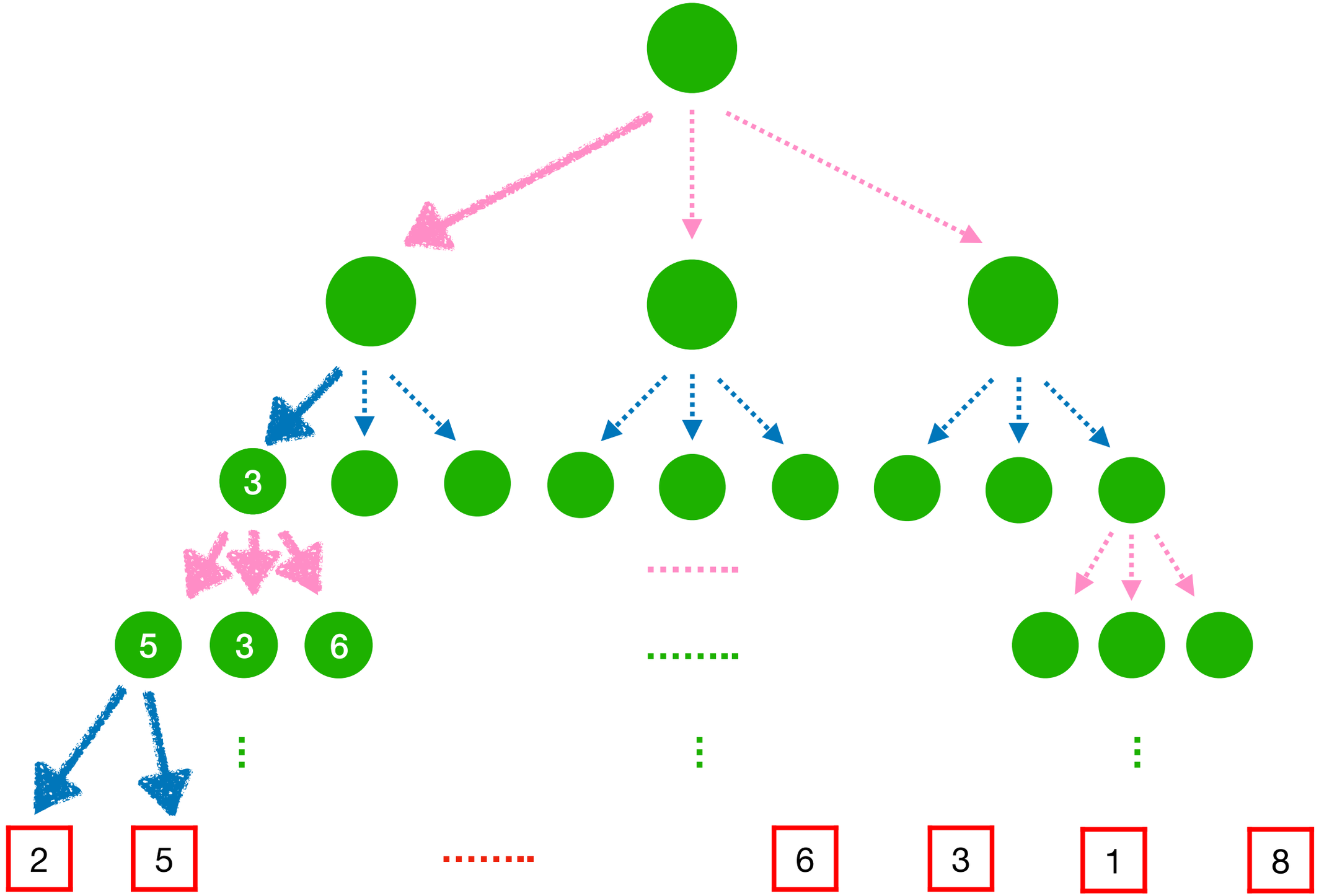
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Tic-tac-toe



Minimax Search



Minimax Search

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

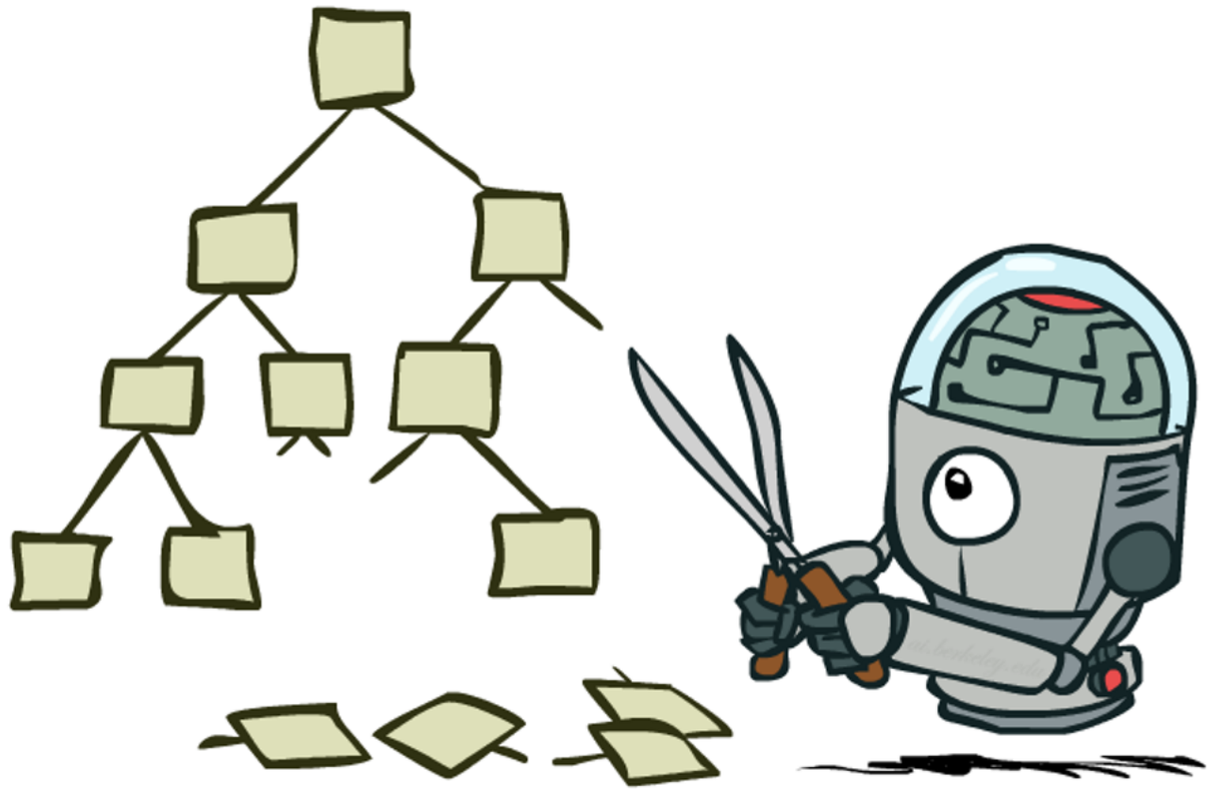
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Works quite similar to depth-first search:
Search down then trace back
from left to right
Similar time and space complexity to DFS

Can We Search More Efficiently?

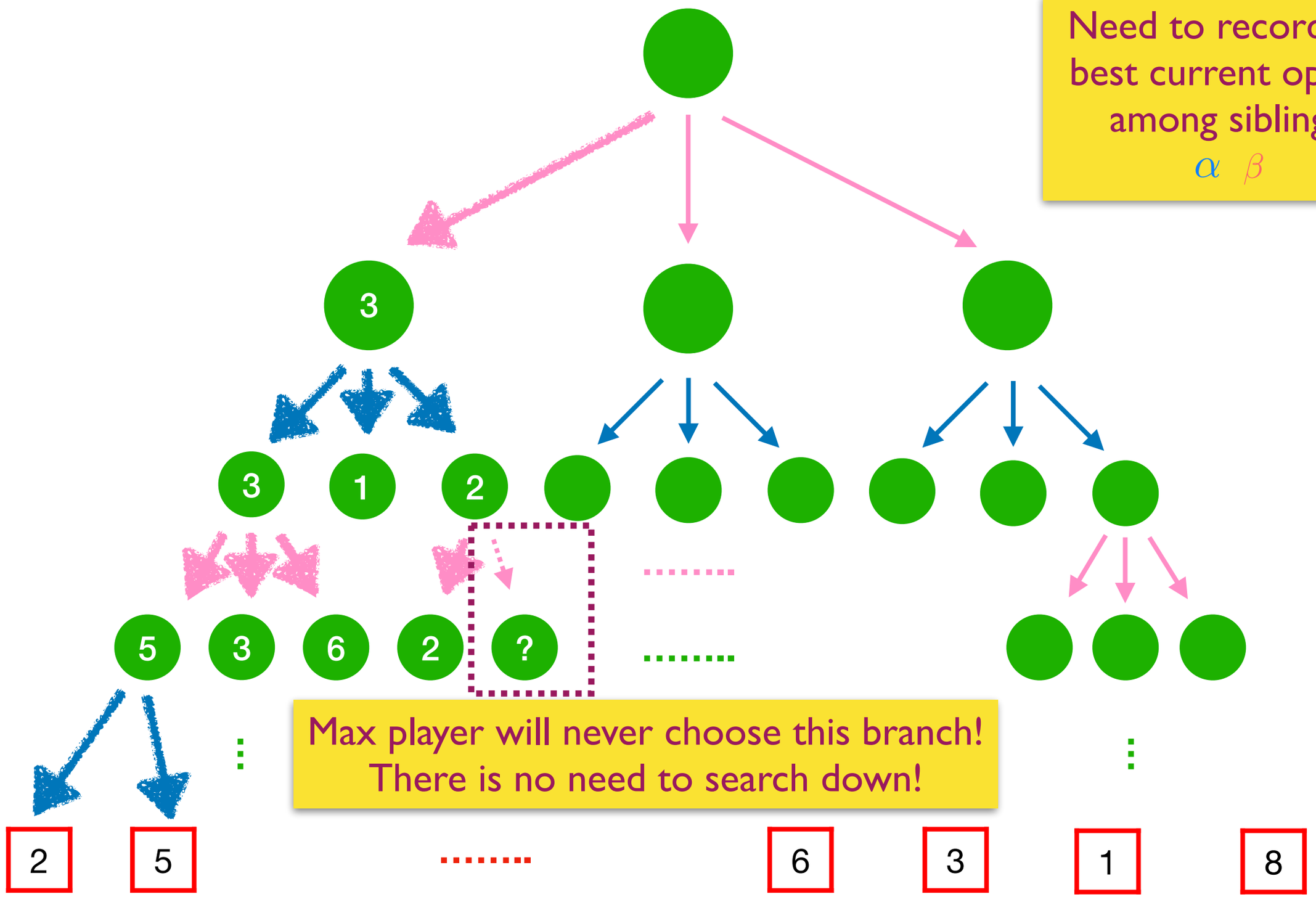
- Limit the search depth and use heuristic functions to estimate the finally utility: e.g. how far away is the ghost?

- Prune the search tree.

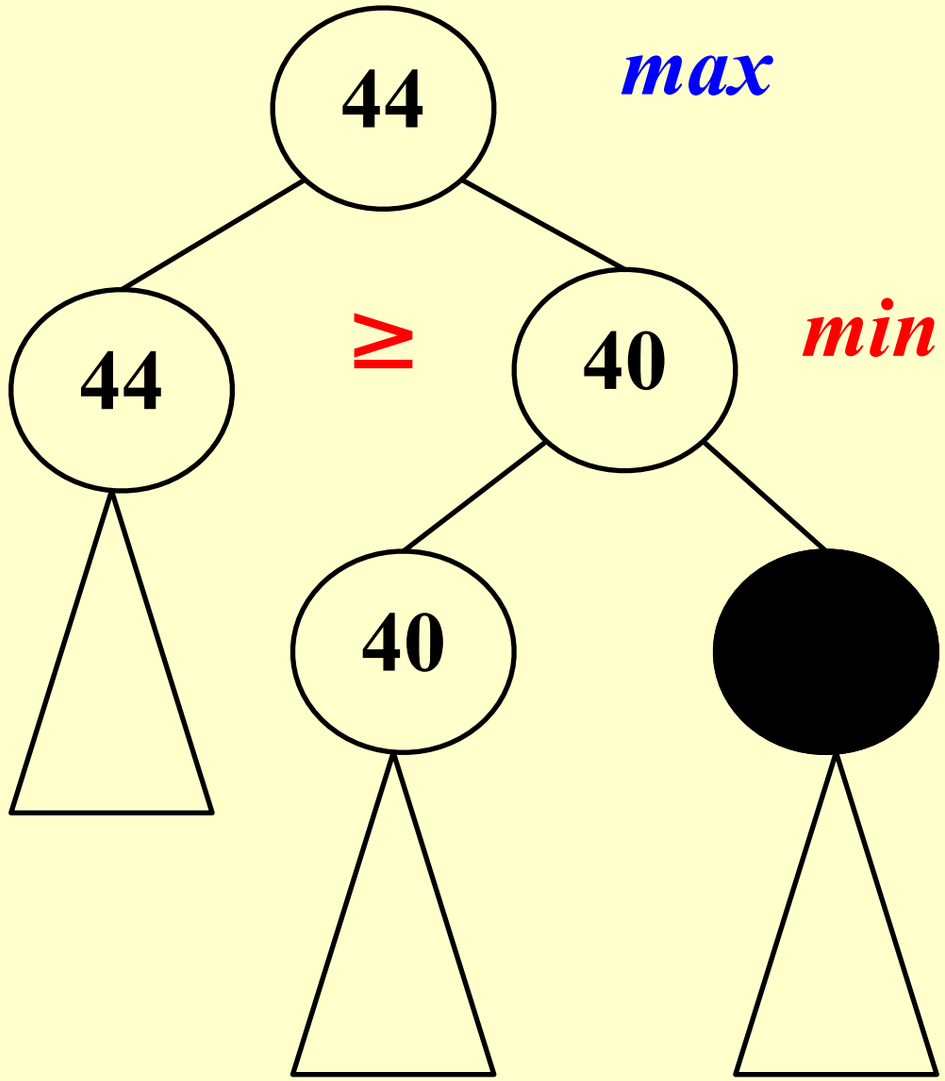


Alpha-Beta Pruning

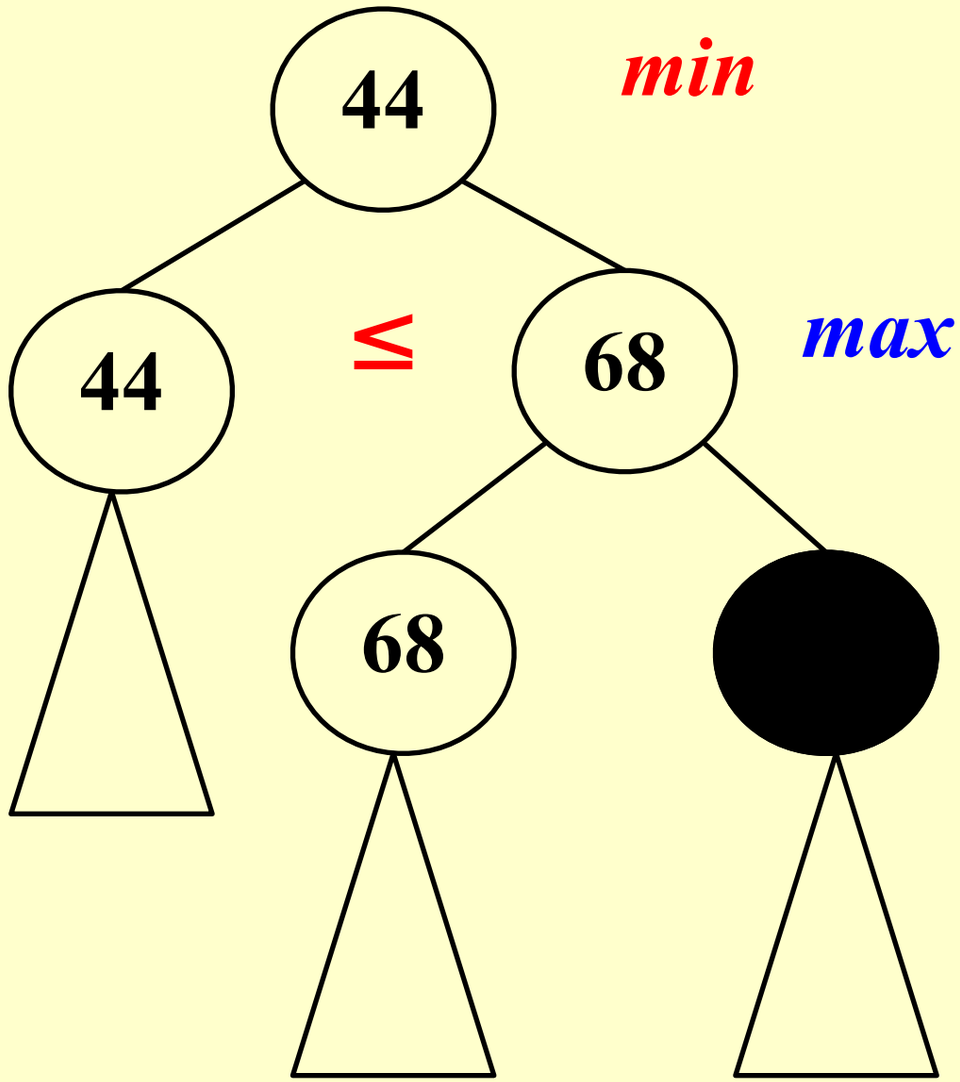
Need to record the best current option among siblings:
α β



α pruning



β pruning



***α-β* pruning:** **when both techniques are combined. In practice, it limits the searching to only $O(\sqrt{N})$ nodes, where N is the size of the full game tree.**

Alpha-Beta Pruning

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

return since the min father will not choose this branch already

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

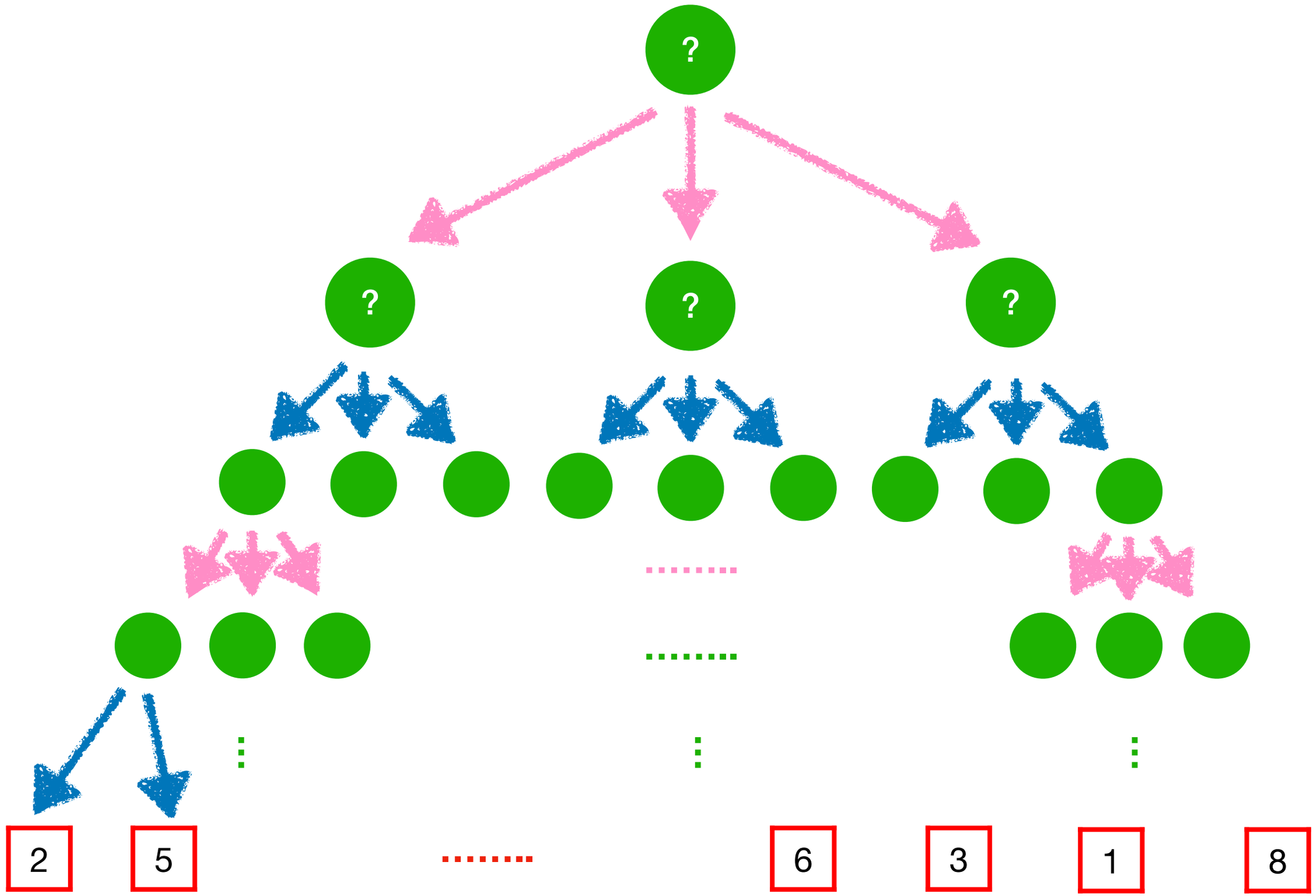
return since the max father will not choose this branch already

Search order matters!
In the worst case, no pruning will be done with bad order!
Deterministic search is not aware of tree structure.
Can we search more smartly?

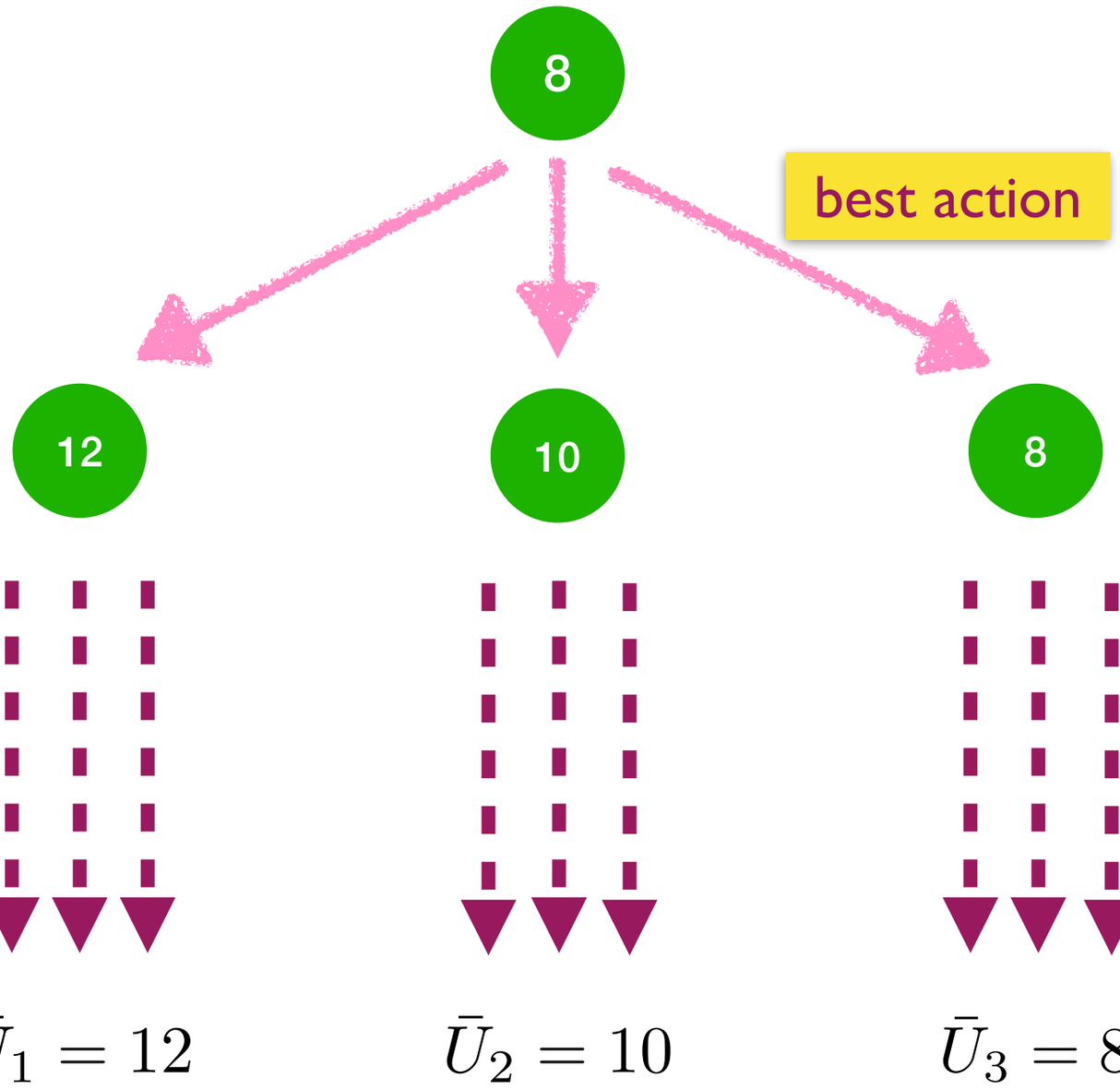
Search & Backtracking

- Fundamental search
- Search with backtracking
- Search by simulation
- Take-home messages

Suppose that the search tree is fully expanded, how to evaluate each mode?



Monte-Carlo Simulation



repeated random run

Converge to true utility when the #run is sufficient!
But in real game playing, the time and space for simulation is limited.
We need a smart strategy to decide the order of simulation.

The Order of Simulation Matters

Chicken & Egg problem:
How to know which is promising when not simulated?

promising branch to simulate more

?

1

5

20

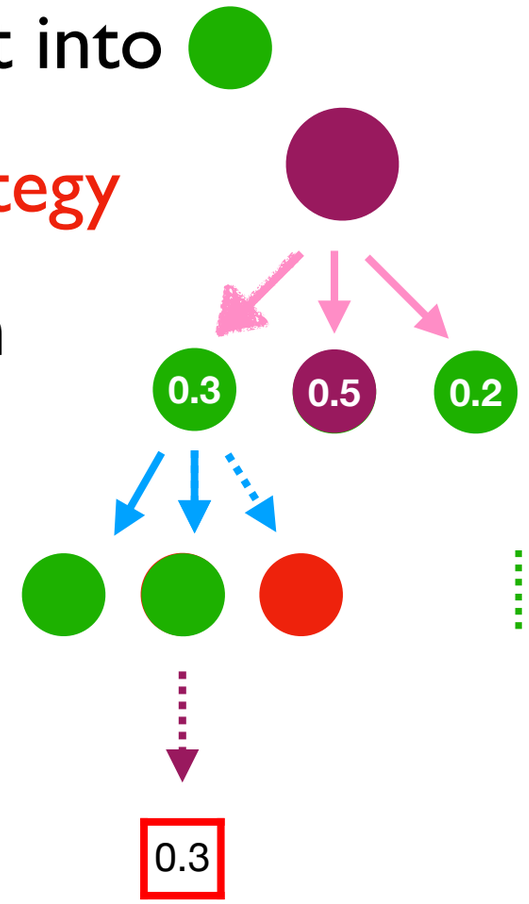
30

15

7

Monte-Carlo Tree Search (MCTS)

- Nodes of two kinds: ● visited node with unexpanded child, and ● unvisited node.
- During MCTS, we use two strategies: tree and default
- Algorithm: repeat until time or space limit:
  - Selection: choose one node among ● using tree strategy
  - Expansion: expand an unvisited child and put into ●
  - Simulation: simulate down using default strategy
  - Update: update MC estimation through path
- Output the best action to play



The default strategy is usually random play
The tree strategy is essential:
Deciding the order of search

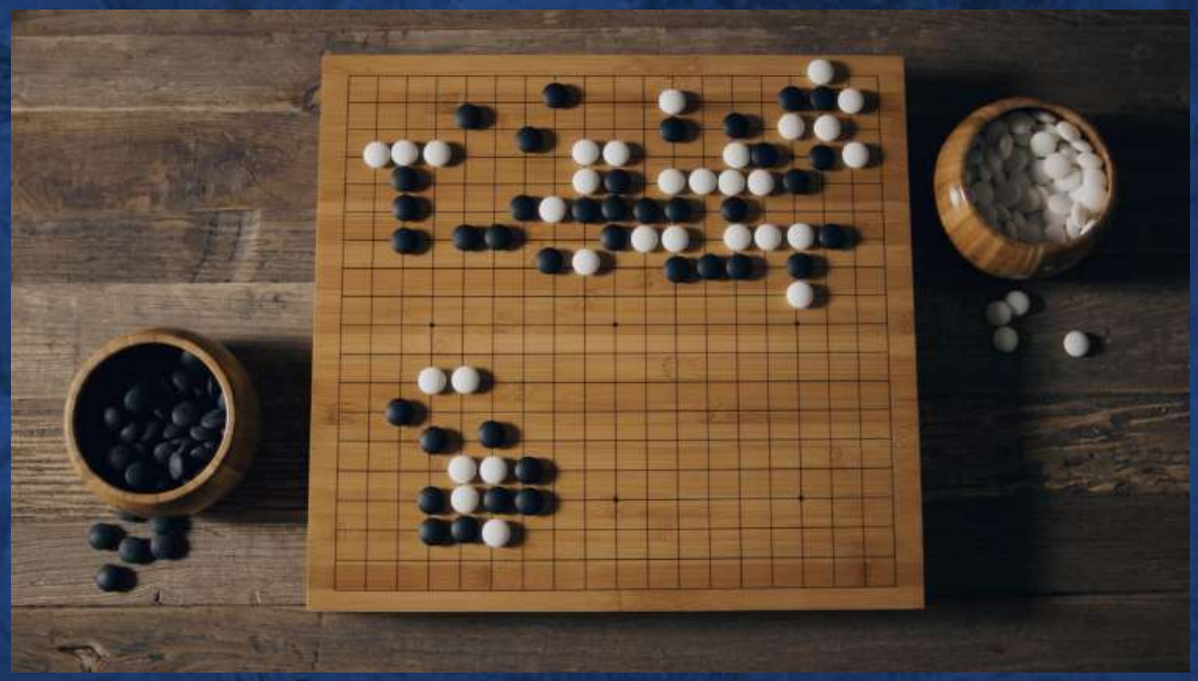
AlphaGo: Integration of Machine Learning & Tree Search

Why is Go hard for computers to play?

Game tree complexity = b^d

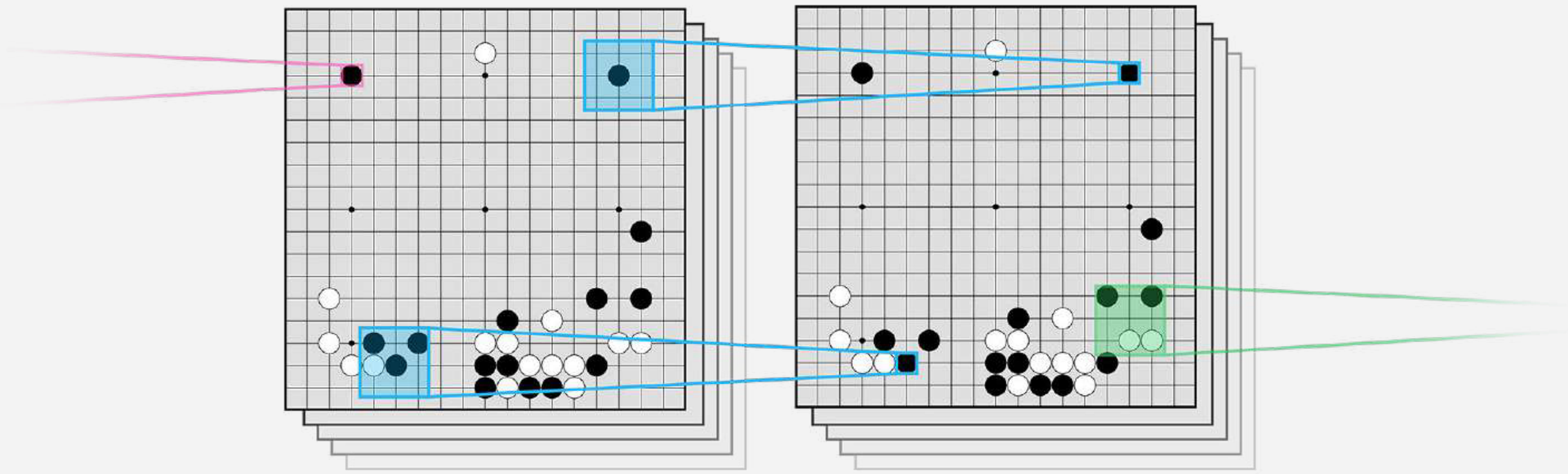
Brute force search intractable:

1. Search space is huge
2. “Impossible” for computers to evaluate who is winning



Google DeepMind

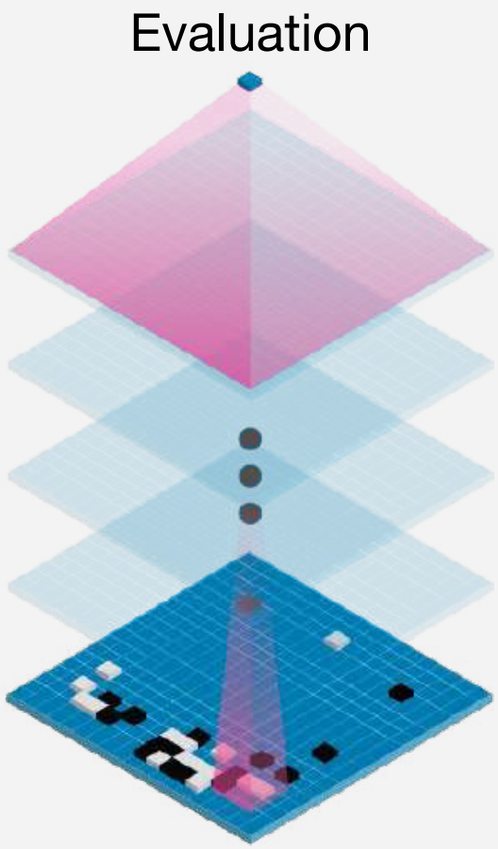
Convolutional neural network

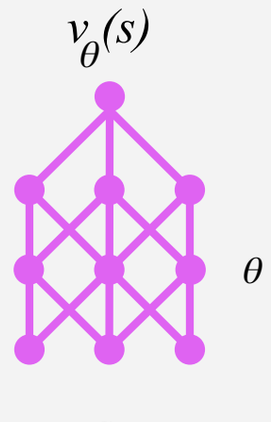


Google DeepMind

AlphaGo introduces three convolutional nets for learning.
Use images of board as state input.

Value network



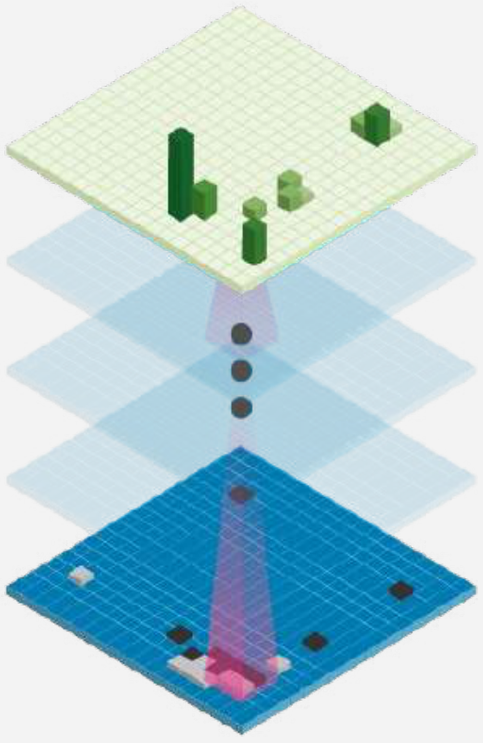


Google DeepMind

The value function evaluates a state of the game.

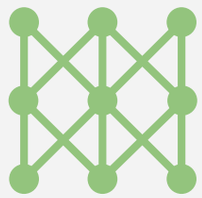
Policy network

Move probabilities



Position

$p_\sigma(a|s)$



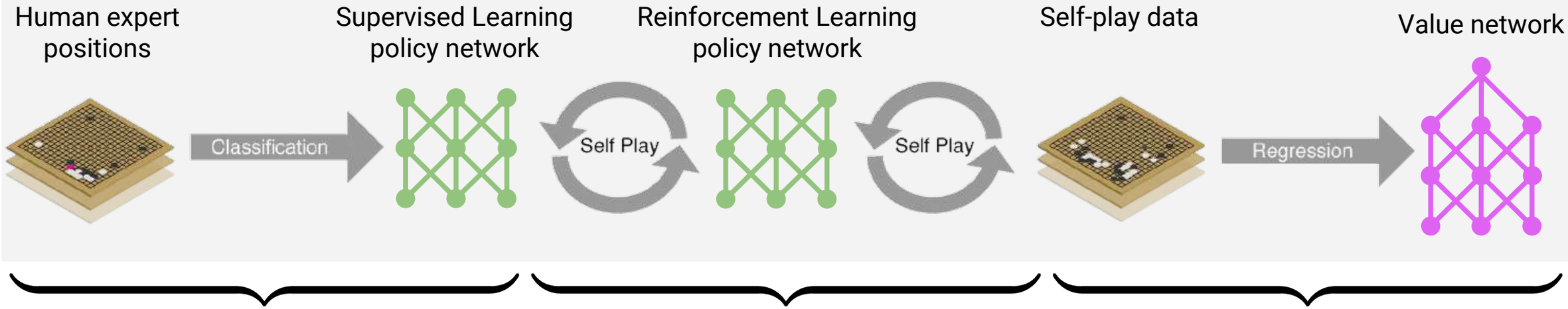
σ

s

Google DeepMind

The policy predicts which is better for the next step

Neural network training pipeline



First stage: supervised classification

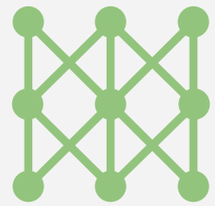
Second stage: reinforcement learning

Second stage: supervised regression

Learning: First Stage

Supervised learning of policy networks

Policy network: 12 layer convolutional neural network



Training data: 30M positions from human expert games (KGS 5+ dan)

Training algorithm: maximise likelihood by stochastic gradient descent

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

Training time: 4 weeks on 50 GPUs using Google Cloud

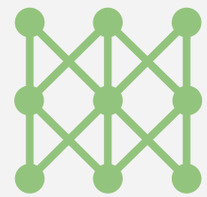
Results: 57% accuracy on held out test data (state-of-the art was 44%)

Google DeepMind

Learning: Second Stage

Reinforcement learning of policy networks

Policy network: 12 layer convolutional neural network



Training data: games of self-play between policy network

Training algorithm: maximise wins z by policy gradient reinforcement learning

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma} z$$

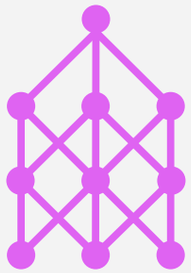
Training time: 1 week on 50 GPUs using Google Cloud

Results: 80% vs supervised learning. Raw network ~3 amateur dan.

Google DeepMind

Learning: Third Stage

Reinforcement learning of value networks



Value network: 12 layer convolutional neural network

Training data: 30 million games of self-play

Training algorithm: minimise MSE by stochastic gradient descent

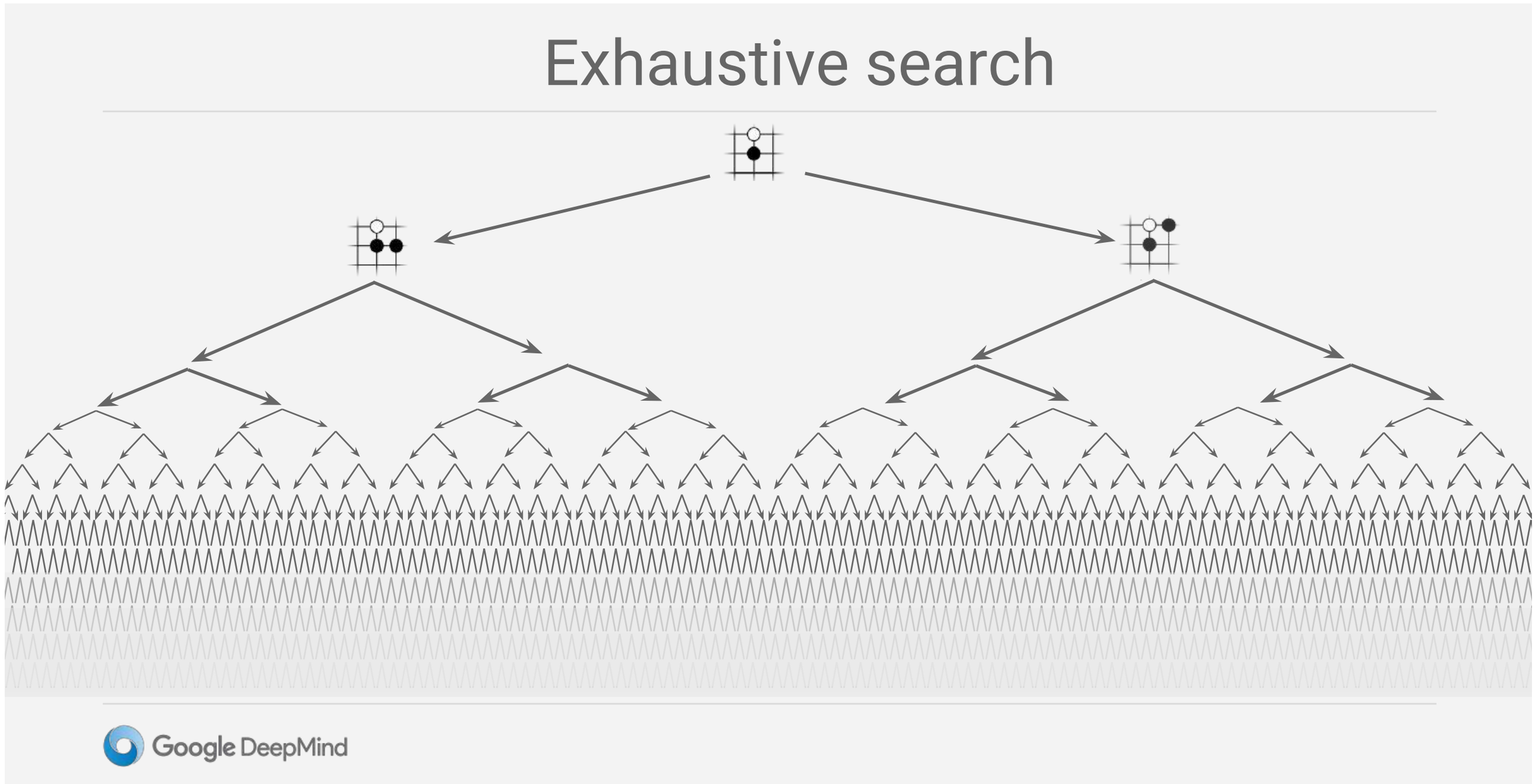
$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$$

Training time: 1 week on 50 GPUs using Google Cloud

Results: First strong position evaluation function - previously thought impossible

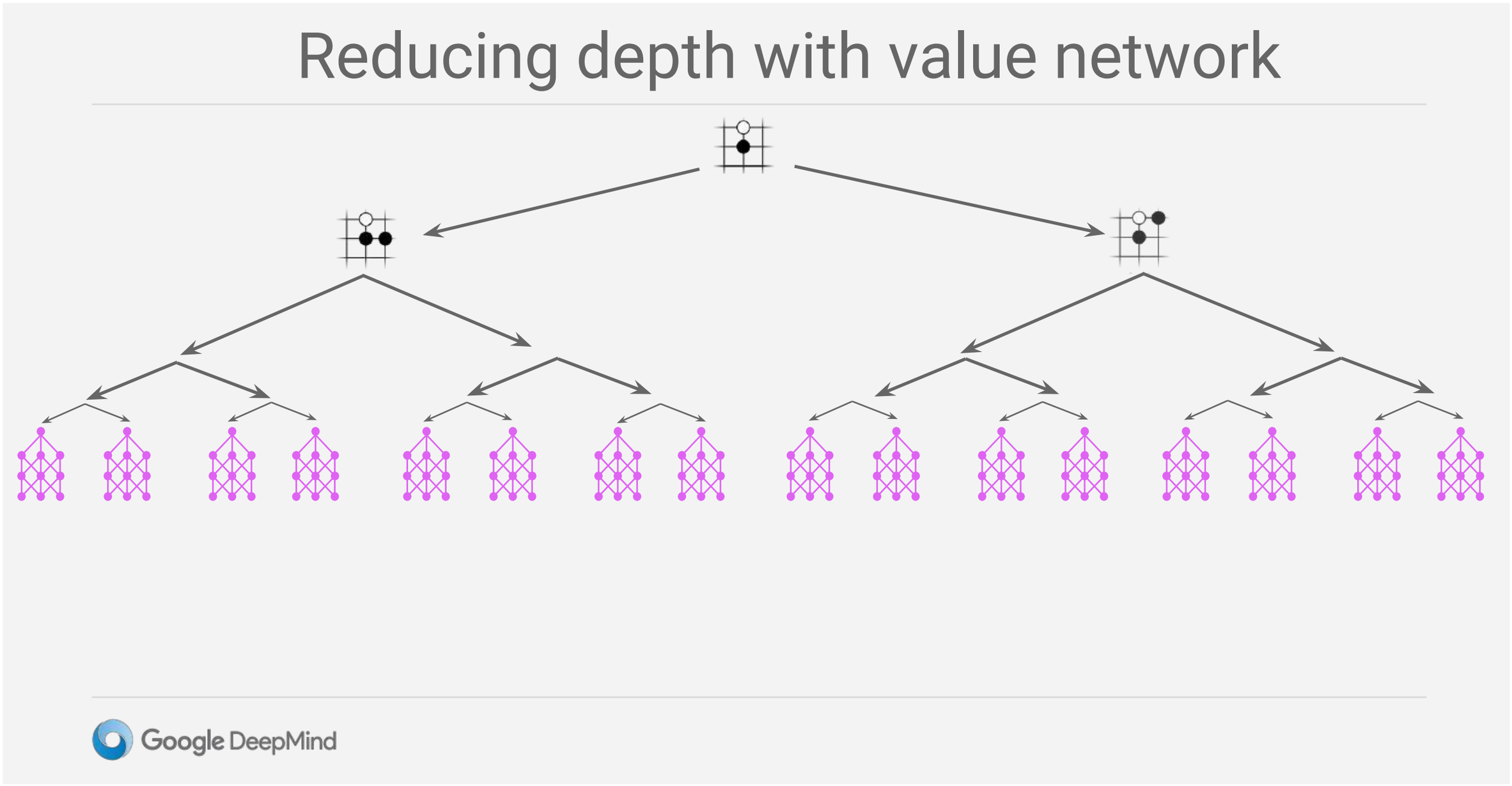
Google DeepMind

Real Play: MCTS



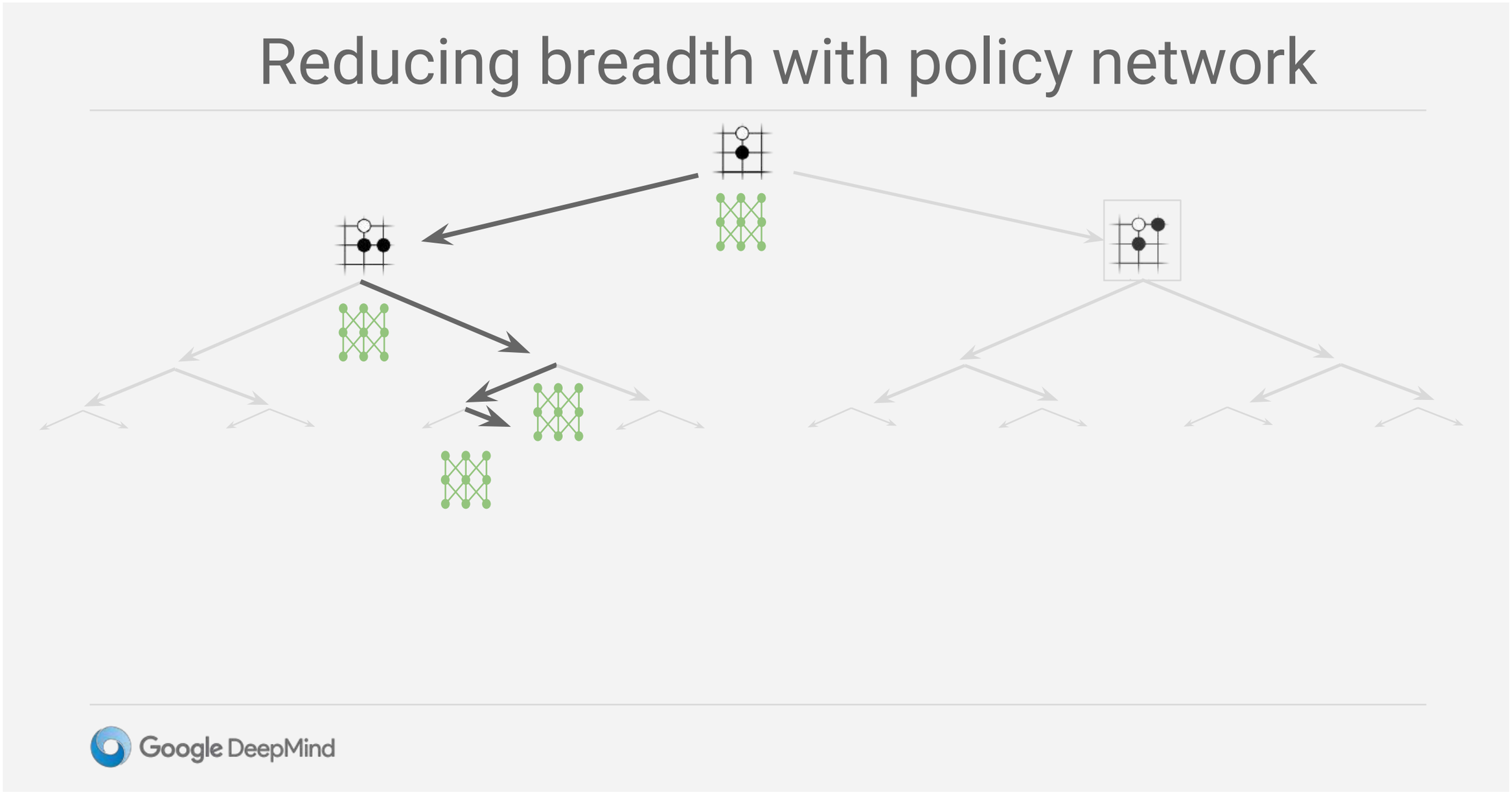
Two key steps:
node expansion and repeated random simulation

Real Play: MCTS



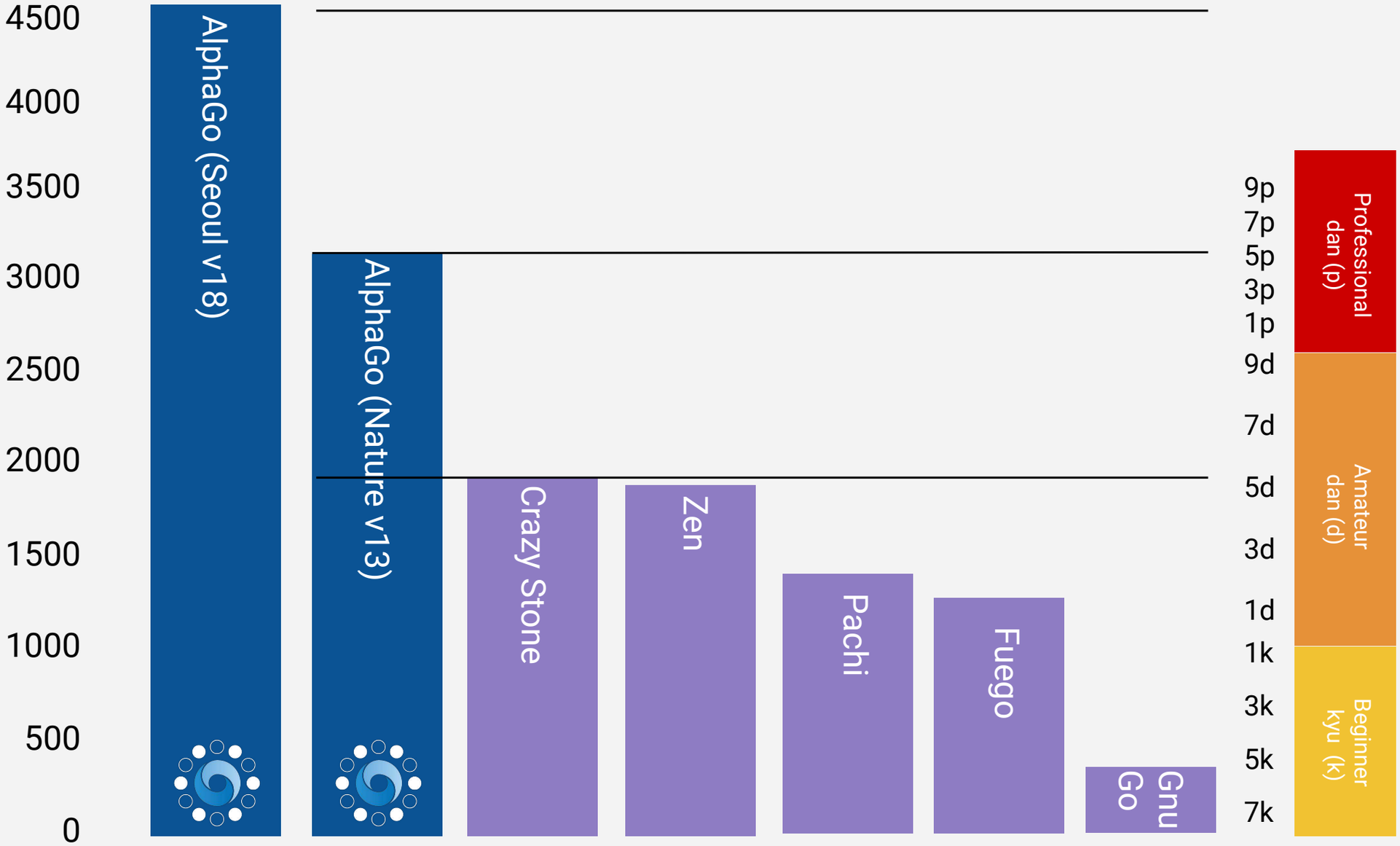
With value network, we can expand fewer depth
since the value of nodes can also be obtained from the value network.

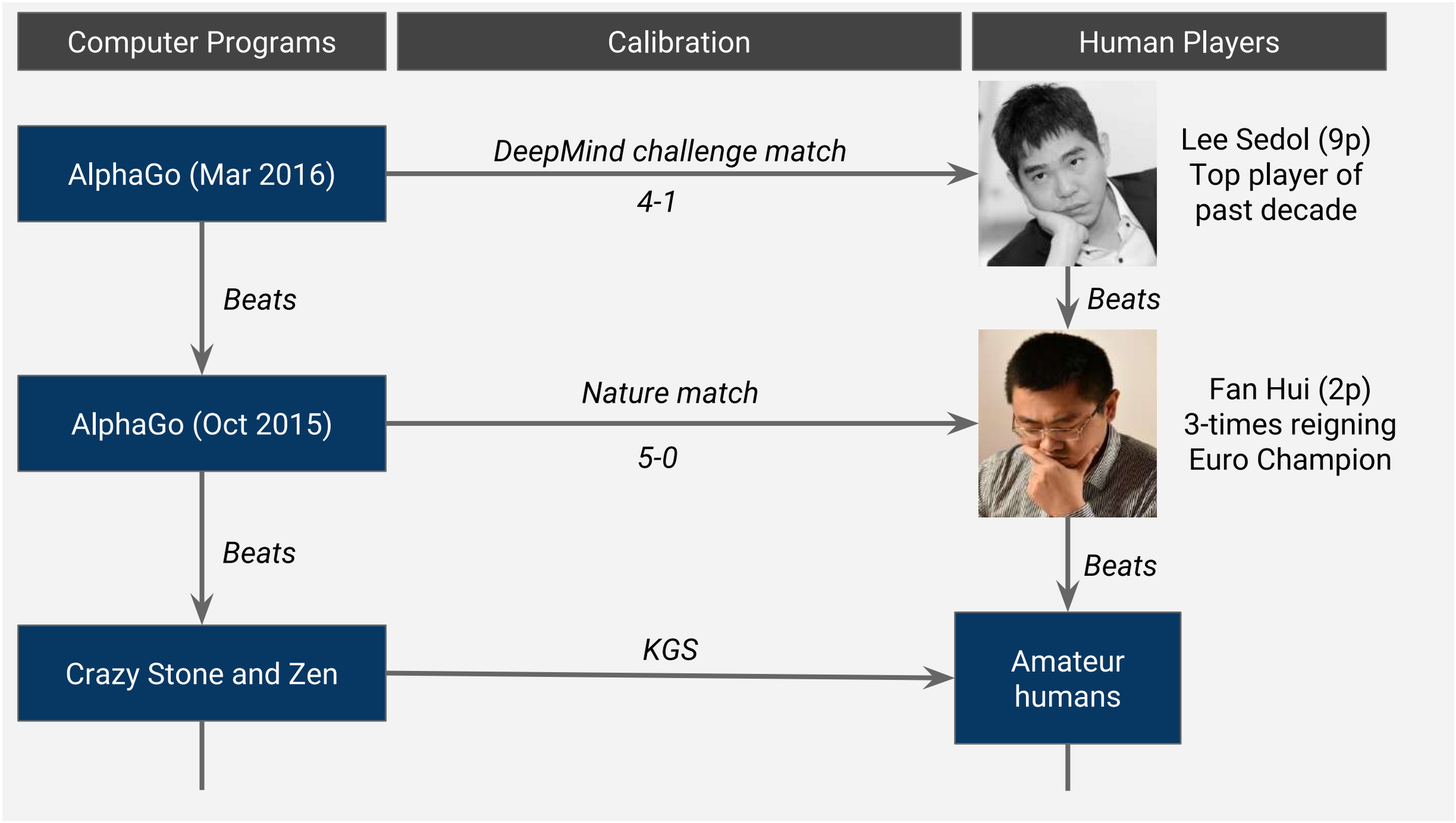
Real Play: MCTS



With policy network, we can simulate with fewer times
since the simulation can be guided by policy network over random play.

Evaluating AlphaGo against computers




Computer Programs
Calibration
Human Players
AlphaGo (Mar 2016)
DeepMind challenge match
4-1
Lee Sedol (9p)
Top player of
past decade
Beats
AlphaGo (Oct 2015)
Nature match
5-0
Fan Hui (2p)
3-times reigning
Euro Champion
Beats
Beats
Crazy Stone and Zen
KGS
Amateur
humans

Search & Backtracking

- Fundamental search
- Search with backtracking
- Search by simulation
- Take-home messages

Search & Backtracking

- Backtracking: search + pruning w.r.t. problem constraints.

- Minimax search and alpha-beta pruning: deterministic search without benefiting from the structure of the search tree.

- Search by simulation: (*optional to learn)
  - Monte-Carlo tree search: benefit from exploration of tree structure and random simulation.

Thanks for your attention!
Discussions?

Reference

Data Structure and Algorithm Analysis in C (2nd Edition)： Sec. 10.5.

Artificial Intelligence: A Modern Approach (4th edition): Chap. 3, Sec.5.1-5.4.

http://ai.berkeley.edu/lecture_slides.html (Lecture 2, 3, 6)